(12)

unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER 86-06-03	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Declarative Descriptions for VLSI Generators		5. TYPE OF REPORT & PERIOD COVERED Technical, interim 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Meei-Chiueh Y. Liem		8. CONTRACT OR GRANT NUMBER(s) MDA903-85-K0072 ARPA-4563, #2, code 5D30.
9. PERFORMING ORGANIZATION NAME AND ADDRESS UW/NW VLSI Consortium, Dept. of Computer Science FR-35, University of Washington, Seattle, 98195		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS DARPA - IPTO 1400 Wilson Boulevard Arlington, Virginia 22209		12. REPORT DATE June, 1986 13. NUMBER OF PAGES 108
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) ONR University of Washington 315 University District Building 1107 NE 45th St., JD-16, Seattle, WA 98195		15. SECURITY CLASS. (of this report) unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution of this report is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

DTIC ELECTE JUL 24 1986 S D E

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

VLSI, VLSI Generators, CIF, DRC, CFL, Caesar

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This paper presents a declarative generator model which can be used to guide the generation of a circuit in VLSI design. The descriptions in the model provide four equivalent representations of a circuit. They are correlated through corresponding components. The syntax and semantics of the declarative descriptions are introduced. Their applications are illustrated by examples. The declarative descriptions are shown to greatly facilitate the VLSI design process and serve as a comprehensive tool that documents the designer's ideas as well as the complexities of a circuit.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-LF-014-6601

unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

AD-A169 952

DTIC FILE COPY

86 7 23 345

Declarative Descriptions for VLSI Generators

Meei-Chiueh Y. Liem

University of Washington
Seattle, WA 98195

Technical Report 86-06-03
June 1986

Accession For
NTIS GRA&I
DTIC TAB
Unannounced
Justification
By
Distribution/
Availability Codes
Dist | Avail and/or Special
A-1

DTIC
COPY
INSPECTED
3

Declarative Descriptions for VLSI Generators

by

MEEI-CHIUEH Y. LIEM

A thesis submitted in partial fulfillment

of the requirements for the degree of

Master of Science

University of Washington

1986

Approved By_____

(Chairperson of Supervisory Committee)

Program Authorized to Offer Degree *Computer Science Department*

Date *June 12, 1986*

Master's Thesis

In presenting this thesis in partial fulfillment of the requirements for a Master's degree at the University of Washington, I agree that the Library shall make its copies freely available for inspection. I further agree that extensive copying of this thesis is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Any other reproduction for any purposes or by any means shall not be allowed without my written permission.

Signature_____

Date *June 12, 1986*

University of Washington

Abstract

DECLARATIVE DESCRIPTIONS FOR VLSI GENERATORS

by Meei-chiueh Y. Liem

Chairperson of the Supervisory Committee: Professor Jean-Loup Baer
Computer Science Department

This thesis presents a declarative generator model which can be used to guide the generation of a circuit in VLSI design. The descriptions in the model provide four equivalent representations of a circuit. They are the layout description, the mixed mode description, the schematic description, and the functional description. Each serves a unique purpose in the VLSI design process. They are correlated through corresponding components. These descriptions are declarative, abstract, robust, and structured in a hierarchical manner. The syntax and semantics of the declarative descriptions are introduced. Their applications are illustrated by examples. The declarative descriptions are shown to greatly facilitate the VLSI design process and serve as a comprehensive tool that documents the designer's ideas as well as the complexities of a circuit.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

I express here my sincere gratitude to Professor Jean-Loup Baer and Professor Lawrence Snyder. I am deeply indebted to Professor Baer for his guidance, assistance, patience, and encouragement. He introduced me to VLSI which is a field I found most challenging and fascinating. He is the major driving force throughout the course of this thesis research. I am also deeply indebted to Professor Snyder for his numerous invaluable ideas, guidance, and inspiration. He has been most instrumental in shaping the major topics of this work.

Dr. Larry McMurchie has been most kind and helpful in introducing me to the intricacies of VLSI tools. He carefully read this thesis and corrected my grammatical errors. This is very much appreciated. I am grateful to Wayne Winder for sharing with me his design on the multiplier which is presented here for illustrating the application of this thesis. My colleagues, Chyan Yang and Wen-Hann Wang, were always there to unfreeze the keyboard and unclog my idea generation process. Their help is appreciated. Thanks are also due to the VLSI Consortium members who provided me the invaluable constructive criticism on this work.

I am most grateful to my husband, Ronnie, for providing me the devotion and encouragement without which this master's program could not have been completed. Lastly, I would like to thank my parents (all four of them) who have given me support and understanding all these years.

CHAPTER 1. INTRODUCTION

VLSI design is an inherently complex process. This thesis presents the development of a declarative generator model which can be used to guide the generation process of a circuit. The description in the model is robust, natural, simple, expressive, abstract, and is structured in a hierarchical manner. It can precisely describe a circuit across its multiple representations at the optimal level of abstraction. It also serves as a comprehensive tool that documents the designer's ideas as well as the complexities of the circuit. This thesis is developed in the context of the VLSI generator project in progress at the University of Washington.

1.1 Description of the VLSI Generator Project

"Quality VLSI Design Generators" is a research project conducted by the University of Washington/Northwest VLSI Consortium. A *design generator* is defined as *a program that produces a family of circuit designs, each one solving a different instance of a particular problem. The input is a problem-specific set of parameters; the output is, among other things, a CIF (CalTech Intermediate Form) definition of the layout of the mask layers* [UW/NW 84]. The objectives of the design generator research are as follows [UW/NW 84]:

1. Build a set of generators that produce quality circuits and which form a complete set with respect to some application. A *quality* design is a robust design that will operate well over a wide range of conditions and that can be consistantly produced.
2. Develop a generator construction methodology with appropriate abstractions, procedures and tools to assure production of correct, quality parts.
3. Demonstrate the efficacy of the approach on substantial designs.

This thesis shows the efforts made towards a generator construction methodology.

The primary components of a design generator are as follows [UW/NW 84].

(1) *parameter validation*: establish that parameters are in the acceptable range of values and possibly "optimize" them if the inputs include, for example, logic equations.

(2) *model construction*: "grow" an abstract version of the circuit. The term "model" refers to a complex data structure that guides the generation process. It includes a logical circuit description, schematic structure of the layout, a catalog of design characteristics (e.g. size, power requirements, etc.) and all the other information needed to synthesize the design. The model fills the gap between the "high level" input and various outputs such as the layout.

(3) *load balancing*: customize the various constituents of the design to the particular situation, e.g. set channel widths.

(4) *generate, using the model*: layout, Design Rule Checking (DRC) interface, ERC interface, simulator interface, etc.

It is hoped that standard generator procedures can be developed and used in a variety of generators. The role that the model plays in the generation process is illustrated in Figure 1-1. In the user frontend interface, the software routines allow the user to specify what he/she wants, such as test to see if the parameter is in the acceptable range, build the model, perform the transformations on the parameters if the input can be optimized, etc. The model is an overall static description of one instance of a circuit. It consists of leaf cells, a set of descriptions, and a catalog which includes the appropriate characteristics of this circuit. These components are produced by execution of the software routines. Under the guidance of the model, the generator routines can generate the layout or the schematic diagrams at the gate or the transistor level of the circuit. The DRC interface, simulator interface etc. are also obtained.

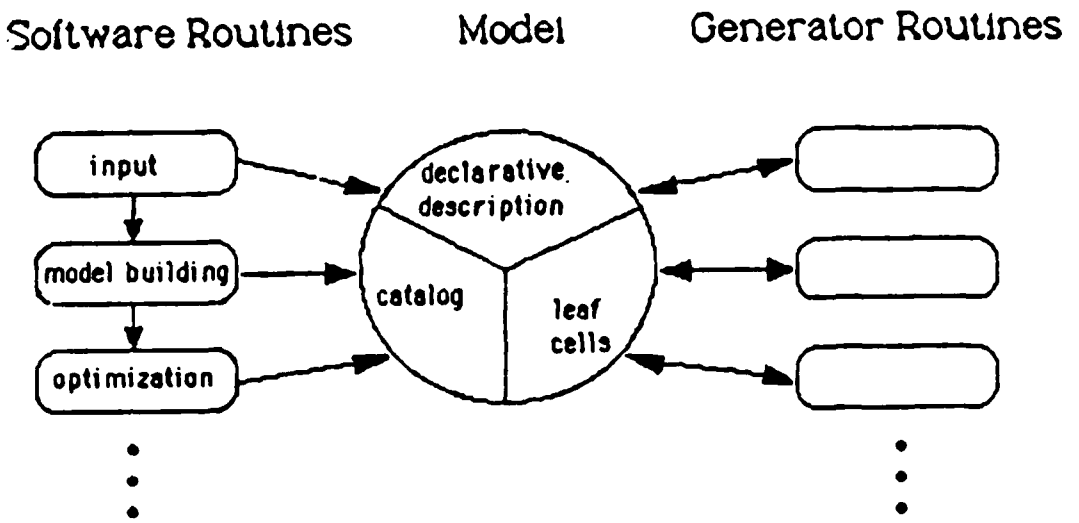


Figure 1-1: The generation process

The model should be sufficiently complete, so that a program which is guided by the model can generate the layout or other circuit descriptions. On the other hand, it should be sufficiently abstract to be able to capture the complex data structure of a design. This thesis concentrates on the declarative description of the model.

1.2 Multiple Representation Problem

The strategy used for building a generator is an implementation of the **divide and conquer** paradigm. After a complex design is partitioned into small modules, each module is defined with a high level description (text, block diagram or schematics) according to the specifications. Then a network of transistors is created and simulated. This step may require a number of iterations to assure the design is stable. When the transistor network behaves as expected, the design is implemented as a collection of integrated circuit layers. It should be noted that in order to build a circuit, the designer goes through a complex, iterative but complete, problem solving process. During this process, the designer needs to constantly interact with the "information base," which may consist of a high level

description, a transistor diagram, or a collection of integrated circuit layouts. These multiple representations of the same circuit must be systematically organized and immediately available to the designer to reduce the complexity of the work. Moreover, in order to design successful circuits, it is important to have a description which not only represents the complexity of the circuits but also allows the designer to express his ideas and design decisions. Without an adequate description of the generator, it is difficult for the designer to build the circuit, check the accuracy of his design and communicate his ideas to others.

However, this facility is not found in the current tool set. The functional descriptions and schematic diagrams are not well documented on line. A generator which creates the layout of a circuit is basically a C program with a lot of Coordinate Free Lap (CFL) calls. CFL is a library of C procedures mainly intended to be used for assembling CAESAR formatted cells[1] into modules. A detailed description of CFL can be found in [UW/NW 85]. Some of the important features of CFL are cited below:

- The system is organized algebraically in that there is a data type called **SYMBOL**, a set of primitive operands of this type, and a set of operators which generate new **SYMBOLs** by forming combinations of existing **SYMBOLs**.
- Box and label are two primitive symbols.
- The operators can be grouped into six classes:

  alignment operators
  linear transformations
  array constructors
  tiling operators
  library access operators
  miscellaneous operators

---

[1]Lower level cells or tiles are generated by the graphical editor CAESAR

- There are two types of routing facilities available in CFL: planar routers and non-planar routers. Each router is specialized to a particular routing situation.
- CFL has two groups of macros -- technology independent macros and technology dependent macros, which can be used to generate frequently used structures such as contacts.
- Wire facility is provided to allow the use of symbol relative coordinates, which is helpful in routing.

Although CFL greatly facilitates the construction of generators, the C program is not precise and expressive enough to capture the hierarchical structure of the circuit layout. When a design is changed, the efforts in modifying the program are not trivial. Therefore, a special need exists for high level descriptions which can precisely describe a circuit across its multiple equivalent representations at the optimal level of abstraction. With this abstraction, the designer can better understand the data structure that guides the generation process. The abstraction will capture the "knowledge" of how an instance of the circuit is built and how the circuits vary with the parameters. Furthermore, it permits us to identify components of the circuit which are independent of the parameters. The isolation of this type of components can reduce the complexity of the design space, which is important in the verification of the correctness of a circuit over the entire space of parameters. Such a high level description will be used to describe the multiple representations of a circuit in the model. The multiple representations will include layout, schematic diagram (gate level and transistor level), and functional description. They must have some acceptable degree of correspondence to better maintain their correctness. Thus the descriptions in the model will serve two functions: (1) design guide, and (2) documentation.

1.3 Survey of Languages for Design and Documentation

This section will survey some research in design languages and information management for VLSI design. Recent major efforts are reviewed here.

Design descriptions are an integral part of the design process. A number of design languages have been published in the literature. According to German & Lieberherr [German 85], hardware description languages (HDLs) can be divided into three categories:

1. Languages that are purely functional specifications and do not necessarily imply a specific structure of the described circuits.
2. Languages that allow both functional and structural specifications. They can be further divided into procedural and non-procedural classes. The latter offers safer descriptions than the former in the sense that more compile-time checks can be done.
3. Languages that are only concerned with structure.

CFL is in category 3. The rest of this section will review languages in each category.

Sheeran [Sheeran 83] proposes a structured hierarchical design language, μ FP (a variation of the Functional Programming language FP), to describe both the semantics (behavior) of a circuit and its layout (a floor-plan). Strictly speaking, it belongs to category 1. The descriptions of μ FP are expressions, made from a small number of primitive functions (functions for manipulating sequences, arithmetic functions and predicates) and combining forms (functions that map functions into functions). These functions and combining forms were chosen because of their algebraic properties. Since each combining form has a simple geometric interpretation, every μ FP expression has an associated floor-plan. It is claimed that circuit descriptions can be easily manipulated using the algebraic laws of the language.

ALI, Zeus and concurrent Prolog are examples in category 2. Lipton, North, Sedewick, Valdes and Vijayan [Lipton 82] use ALI, a procedural language, to specify VLSI layouts. The main feature of ALI is that it allows its user to design layouts at a *conceptual level* in which neither sizes nor positions of layout components may be specified. A layout is regarded as a collection of *rectangular objects* and a set of *relations* (primitive operations, such as *above, glueright, inside,* etc.). One result generated by an ALI program is a set of linear inequalities that embody the relations between the layout elements. These inequalities are then solved to generate the positions and sizes of the layout component. Since the design rules are incorporated as a table which is used by the primitive operations and *completeness* of the layout descriptions is checked hierarchically, the layout generated is *free of design rule violations* and is relatively easy to update to new design rules. However, ALI can not handle rectangles whose sides are not parallel to the cartesian axes. The use of the cell mechanism creates a certain waste because the minimum separation between cells is the maximum of all the minimum separations for two layers in the design rules. In addition, ALI can not make inferences as to the relations between boxes beyond those implied by the transitivity of some primitive operations; this leaves the user with a fair amount of drudgery to make sure that the program is complete so that the layout will be free of design rule violations.

Zeus [German 85] is an HDL whose principles of structuring and much of the syntax are modeled after MODULA-2; however the semantics are radically different. The notations in Zeus can simultaneously express both the *structure* and *function* of a circuit, and emphasize the design of regular structures in hardware algorithms. Zeus provides facilities for describing circuits by recursive and iterative methods. It allows the designer to specify and prove the functional correctness of entire parameterized families of designs.

Suzuki [Suzuki 85] describes VLSI chips as concurrent building blocks connected by wires. Thus, concurrent Prolog was chosen to write input and output assertions as well as hardware specifications. System components are described hierarchically -- circuits as predicates and connectors as predicate parameters. Since the processes and scheduling are inside the language processor and therefore inaccessible to the programmer, it is hard for the user to write a more sophisticated simulation system.

Examples of languages that are only concerned with structure are **Regular Structure Generator** (RSG), Escher, and SLL. Abstraction mechanisms including *macro abstraction, delayed binding, interface inheritance*, and *the complete decoupling of graphical and procedural design information* are implemented in the RSG [Bamji 85] to provide the designer with the most profitable level of abstraction and make the regular circuit structure generally accessible.

A circuit layout is generated from three input files: *design file, layout file, and parameter file.* Local and global efficiency are achieved by completely decoupling the graphical and procedural domains. The RSG uses previously defined cells to hierarchically build larger cells. By *macro abstraction*, i.e. the specification of macrocells as interconnections of smaller cells whose binding on location and orientation can be *delayed* to any desired time, the designer can concentrate only on the connectivity of the subgraph. The *interface* between two cells is defined as the ordered pair of *interface vector* and *interface orientation. Interface inheritance* provides a powerful means to define interfaces: A new interface between two macrocells can be computed from any legal interface between a subcell in the first macrocell and a subcell in the second. The relative placement of cells in the final layout is performed using an *interface* between cells and not by using the *sizes* and *shapes* of the bounding boxes of those cells. This

makes cell design and design rule check easier. However, decisions based on the size and shape of the final layout such as placement and routing are difficult to make.

Escher, a geometrical layout system for recursively defined circuits, is described by Clarke and Feng [Clarke 85]. An Escher circuit description is a hierarchical structure composed of cells, wires, connectors between wires, and pins that connect wires to cells. Cells may correspond to primitive circuit elements, or they may be defined in terms of lower level subcells. Unlike other geometrical layout systems, a subcell may be an instance of the cell being defined. When such a recursive cell definition is instantiated, the recursion is unwound in a manner reminiscent of the procedure call copy rule in Algol-like programming languages. Cell specifications may have parameters that are used to control the unwinding of recursive cells and to provide for cell families with varying number of pins and other internal components.

SLL, a Symbolic Layout Language [Ellis 81], is the human-readable form of the schematic, logic, layout, and simulation information about a circuit in the Ruby database system used at the University of California, Berkeley. It also serves as a general interchange format for integrated circuit geometric data. Connectivity information and the hierarchy and regularity present in circuits are stated explicitly in SLL. It is claimed that any geometry that can be expressed in CIF can also be expressed in SLL. Cell parameters and special constructs for arrays and busses replace lengthier equivalents in CIF. SLL also allows symbolic naming and arbitrary nesting of cell definitions.

Research in information management for VLSI design has received more and more interest recently. Chu and Lien [Chu 84] describe the VLSI design database (Vdd) system as a set of programs targeted to assist a circuit designer in layout

design, verification, and simulation. There are three goals that the Vdd software should achieve:

- support a layout description language.
- be technology independent as much as feasible.
- be complete in its own right for a layout designer.

The database techniques used include:

- use of a relational schema to describe a silicon processing technology.
- model the chip information by a set of relations, each of them representing a component type.
- swap data in and out of the main memory storage according to data semantics.
- implement the design session as a database transaction.

Katz argues that design data management is one of the most important areas of VLSI design, yet it is also one of the least understood. He discusses a lot of critical issues in information management for VLSI design [Katz 82, Katz 83a, Katz 83b, Katz 85a, Katz 85b]. Some of the important ones are summarized below:

- No existing commercially available system supports the complete range of facilities needed to support design activities. In particular, the features missing include: an explicit representation of the design hierarchy, support for a flexible choice of design representations, and a multi-level architecture.
- A design database should organize the design information across representations, alternative implementations, and evolutionary versions. By making the dependencies among parts of the design explicit, the ramifications of design changes can be more easily discovered and propagated in a controlled manner.
- Designs are organized into a richly interconnected data structure using an *object data model*. Objects can be *representation objects*, *index objects*, *equivalence objects*, *alternative objects*, *version objects* or *generic objects*. Representation objects have interface descriptions specifying their abstract behavior, usage information, and associated performance (speed, power, area).
- A prototype design management system should consist of *storage component*, *object system*, *recovery subsystem*, *design librarian*, *design validation subsystem*, *design transaction*, *tools*, *browser* and *chip assembler*.

1.4 Structure of the Thesis

The remainder of this thesis is structured as follows. In the next chapter, we will provide the notations used for describing a circuit. The description of multiple design representations is illustrated through a simple example, **decoder**, and a more complex design, **multiplier**. Chapter 3 presents parameterization issues involved in the use of notations to describe a circuit. Finally, chapter 4 summarizes the material presented in chapters 2 and 3. The contributions made by this thesis in design and documentation of VLSI circuits and suggestions on further work will also be presented.

CHAPTER 2. DECLARATIVE DESCRIPTIONS

A set of unambiguous notations is needed for the high level descriptions to aid the designer in the design and documentation process. The purpose of this chapter is to introduce a set of notations and show how these notations are used to describe the multiple equivalent representations of a design. Section 2.1 gives the main features of the high level descriptions in the model. Data elements -- *leaf cells* and *abstract objects* -- will be described. The syntax as well as the semantics of operators will also be discussed. Section 2.2 illustrates how the notations are used to describe a simple circuit -- **decoder**. Four types of representations -- *layout*, *mixed mode* (mixture of transistors and gates), *schematic diagram*, and *functional description* -- are examined in detail, to see how design and documentation work can be enhanced by using these notations. Section 2.3 gives a more complex example -- **multiplier** -- to show the versatility of the notations.

2.1 Main Features

2.1.1 Overview

The high level descriptions which will be used for design and documentation in the generator project should have the following fundamental properties.

(1) *Simplicity* and *Naturalness*. Simplicity is an important language design principle. A simple set of notations makes the description easier to read and write. The notations should also be as natural as possible to increase the understandability of the description. A simple correspondence between the geometric notations and the actual placement relationships will allow changes in relative positions to be easily reflected in the changes in the notations.

(2) *Expressiveness*. While the notations should be simple, clear and natural, they should also be sufficiently descriptive to allow designers to fully describe the hardware aspects and other design decisions that a particular instance of a design intended to include. Otherwise, effective and efficient communication between designers is impossible. A very detailed description of a circuit, however, can blind one to its general properties. As a result, the desired expressiveness of the notations is a compromise between these two extremes.

(3) *Abstraction* and *Hierarchical Structure*. In order to reduce the complexity of the VLSI generator design process, hierarchical decomposition and abstraction have to be employed. The circuit eventually has to be specified in terms of the primitive modules before it can be implemented. However, with abstraction the complexity of the circuit is better handled. Designers can specify circuits in increasing order of complexity. Information about lower levels in the hierarchical tree are completely hidden from higher levels. Abstraction makes the data structure explicit.

(4) *Technology Independence*. In response to the rapid development in VLSI circuit research, the notations should be robust across different kinds of technology. In other words, the circuit description of a design using technology X should be the same as that of the same design using technology Y. The only changes that a designer has to make due to the change of technology are the innards of leaf cells, since they are the primitive components of a circuit. This will simplify the conversion of a description to support a new technology.

With these desired properties, the high level description can precisely describe a circuit across its multiple equivalent representations at the optimal level of abstraction. In our research, each description describes an instance of a family of circuit designs in one of four possible representations. A description consists of

two parts: (1) the *declarative part*, which includes the name of a circuit, the type of the representation, a list of parameters, a collection of leaf cells, and a set of imported functions; and (2) the *imperative part*, which begins with the keyword **MAIN**. In this part, a collection of statements is used to describe an instance of a circuit, e.g. a decoder with three select wires in NAND gate style. The description for the layout or the schematic representation can be regarded as a collection of *objects* (leaf cells or abstract objects) and a set of *relations* among these objects. For the functional description, the intermediate hidden mechanisms between inputs and outputs are described.

Free-format input is used in a description. In other words, statements can be positioned anywhere in the input line. This will help avoid syntax errors due to improper positioning of tokens and permit the whole description to be laid out so that it is easy to read. Hierarchical structure can also be shown by indentation. The syntax of this new high level description is designed to be as close to that of the "C" programming language as possible since the generators are written in C. The Extended Backus Naur Formalism (EBNF) definition for the declarative description is given in Appendix A while the rest of this section describes the main features of this description in its four subsections: **declaration, objects, operators and flow of control**.

2.1.2 Declaration

The syntax of the declarative part of a description is of the form:

NAME <circuit_name> ;

TYPE <representation_type> ;

PARAMETER <parameter_list> ;

LEAF CELLS <cell_list> ;

FUNC <function_list> ;

Boldface characters are used to indicate keywords and required syntactic elements. **FUNC** <function_list> is optional and **LEAF CELLS** <cell_list> are not used in the functional description. <representation_type> is either **LAYOUT, MIXED, SCHEMATIC** or **FUNCTIONAL** for layout, mixed mode, schematic diagram or functional description, respectively.

<parameter_list> is a list of *inputs* to the circuit. For example, in the case of a 4 bits by 3 bits multiplier, it is the number of bits in the multiplicand (say, m = 4) and the number of bits in the multiplier (say, n = 3). Thus, the <parameter_list> is **m = 4, n = 3**. The names **m** and **n** are arbitrarily chosen; however, the values that they are bound to in the declarative part are constant through the entire description. Parameter declarations allow implicit dependencies to be made explicit, which makes the modification of the circuit description very easy when the inputs are changed. Suppose that it is decided that the number of bits for the multiplicand should be changed to 8 (say, to accomodate bigger numbers). This will imply that the limits on some of the iterative contructs such as the number of partial product generated should also be changed. The designer can simply change the parameter list without having to find all numbers that implicitly depend on the changed bit size and make appropriate changes. Therefore, parameter declaration enhances the readability and maintainability of a description.

Leaf cell is the lowest level module in the hierarchy of a description. It is a primitive component. More details about leaf cells will be given in the next subsection. <cell_list> contains all the leaf cells that are used in a description to generate a geometric representation (the layout or schematic diagram in gate level or transistor level) of a circuit. The leaf cells should be copied from the system library to the user's working directory. They are unaltered across the family of instances.

Following the <cell_list>, the functions that aid in the circuit description are specified. This part must begin with the keyword **FUNC** followed by a list of functions delimited by commas. Functions are user-defined. For example, *binary* can be a function which will return a binary representation of a number. To specify a circuit in a clean and understandable manner, the implementation parts of functions do not appear in the same file as the circuit description. A function provides a convenient way to encapsulate computations in a black box, which can then be used. In this manner, another language design principle - *information hiding* - is achieved. Functions provide a way to cope with the potential complexity of a large VLSI design description.

The declarative part ends on encountering another keyword **MAIN** which is followed by the imperative part of a description.

2.1.3 Objects

Leaf cells are the *primitive objects* to which geometric operators (discussed in the next subsection) can be applied, and out of which more complex objects (*abstract objects*) can be built in the layout or schematic representations. They are declared explicitly in the declarative part as described in the previous subsection. Leaf cells are global in scope. In general, they have some predefined, sufficiently general, functionality. A leaf cell is a rectangle (also called **box**) with its sides parallel to the axes of a cartesian coordinate system. It can be a NAND gate used for the schematic description of a decoder or a physical layout of a half-adder used for the layout description of a multiplier. The designer has the flexibility of creating whatever leaf cells he needs. The choice of these leaf cells and their suitability will depend on the graphical editor CAESAR, CFL and other software/hardware tradeoffs. A leaf cell encapsulates implementation details such as the layers used to fabricate the layout within a box. The implementation details

are transparent to the outside, and leaf cells are unchanged across the family of instances.

The high level description that is developed here also has the capability of specifying iterative (arraylike) structure. A leaf cell can be *instantiated* as many times as desired by specifying the number of repetitions. For example, if **dec_na_i_inv** is a leaf cell, then **| (dec_na_i_inv (n))** means to create an object which is a collection of n copies of **dec_na_i_inv** and the relations among these copies will be defined by the geometric operator |. Array constructs allow easy specification of regular structures without adding the additional complexity of control flow structures to a description. A leaf cell without an argument is by default an instance of that cell in the user's working directory.

Abstract objects are created to provide designers with the mechanism to describe a circuit representation hierarchically so that most of the details at one level of the hierarchy are truly hidden from all higher levels. An abstract object can be defined recursively. An alias of a leaf cell, an array of leaf cells, a group of heterogeneous leaf cells, or a combination of the last two is an abstract object. Moreover, an array of abstract objects, a group of heterogeneous abstract objects or a combination of these two is also an abstract object. It should be noted that an abstract object represents an integrated consecutive part in the geometric placement. Generally, it is a module which has some functionality. Since leaf cells are represented as rectangles, a simple example of an abstract object can be a "row" of leaf cells. It is obvious that an abstract object can also be instantiated as many times as desired by providing the appropriate arguments. Thus, **--(row[i](i=0..4))** means to generate five rows: row [0], row [1], row [2], row [3], and row [4]. The relations among these rows are defined by the operator --. An element of an array of abstract objects can be accessed by specifying the subscript as in most

programming languages. Since row [i] is an abstract object, information about its lower level description is hidden. At this level of abstraction, row [i] can be thought of as a primitive component. Abstract objects are considered global. Thus, object names within a description must be unique.

Given the features of leaf cells and abstract objects mentioned before, each description can use many levels of abstraction. The circuit is viewed as a network of objects at each level of abstraction. At the highest level of the hierarchy, it is a single abstract object -- the name of the circuit that the designer intends to describe, say, decoder. At the lowest level of abstraction, the circuit is a network of leaf cells. To be more specific, the description of a representation of a circuit is recursive in nature -- each abstract object is specified as a network of lower level objects. The internal details of these lower level objects need not be known when defining an abstract object. That is, the lower level objects can be thought of as primitives at that level of abstraction. Since an abstract object is defined after it is used, the description of an object can be deferred until the design is better understood. The abstract object may itself be a named instance of a higher level object to aid the description of the recursively defined circuit.

2.1.4 Operators

This section shows the operators which are used in our declarative descriptions. They can be arranged in the following groups: (1) *geometric operators*, (2) *arithmetic operators*, (3) *relational operators*, (4) *logical operators*, and (5) *the assignment operator* (=). We will concentrate on the geometric operators in this section.

2.1.4.1 Geometric Operators

As mentioned before, the layout description, the mixed mode description, and the schematic description can be regarded as a collection of rectangular objects and a set of relations among these rectangles. The relations between objects are defined by geometric operators. These operators take objects as arguments and produce objects as results. A small set of geometric operators are created[1]. The first four operators are used to combine objects into more complicated objects, while the last three operators are used for linear transformations. They are described as follows.

(1) -- : *beside*. A -- B denotes that object A is on the left hand side of object B as shown in Figure 2-1.

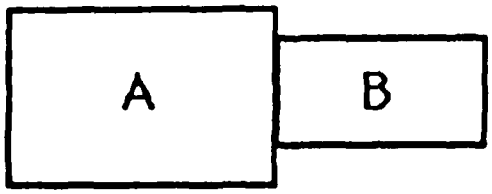


Figure 2-1: A -- B

---

[1]rot, mx, and my are patterned after the same operators in CFL.

(2) | : *above*. A | B means that object B is above object A. The geometric relation is shown in Figure 2-2.

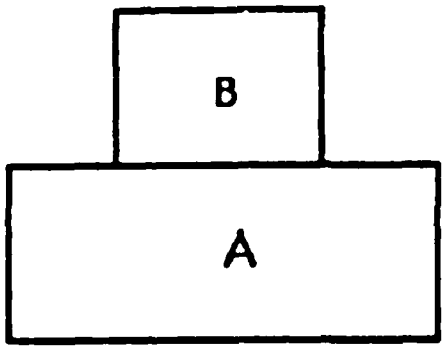


Figure 2-2: A | B

(3) -- ∩ : *horizontally joined with overlap*. A -- ∩ B denotes that object A and object B are horizontally joined and overlapping with A situated on the left hand side of B. Figure 2-3 shows the relations.

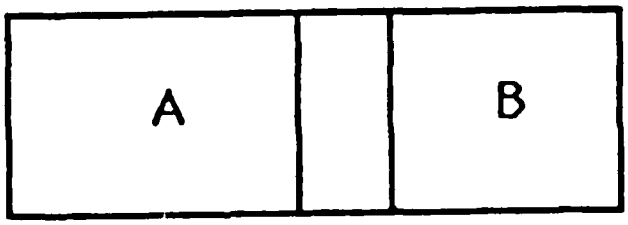


Figure 2-3: A -- ∩ B

(4) $| \cap$: *vertically joined with overlap*. A $| \cap$ B represents that object A and object B are vertically joined and overlapping with B situated on the top of A. The geometric relation is shown in Figure 2-4.

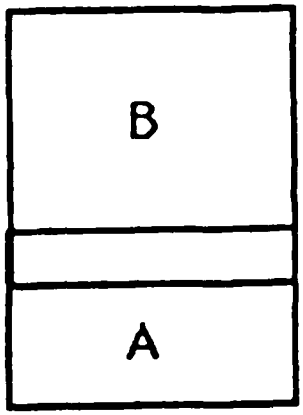


Figure 2-4: A $| \cap$ B

(5) **rot** : *rotate*. rot(A, -90) denotes that object A is rotated 90 degrees clockwise as shown in Figure 2-5.

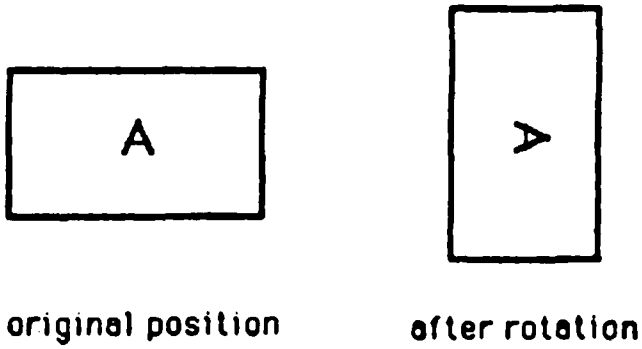


Figure 2-5: A is rotated -90 degree

(6) **mx** : *mirror in x.* $A' = mx(A)$ means that A' is the mirror image of A across the X axis, which is shown in Figure 2-6.

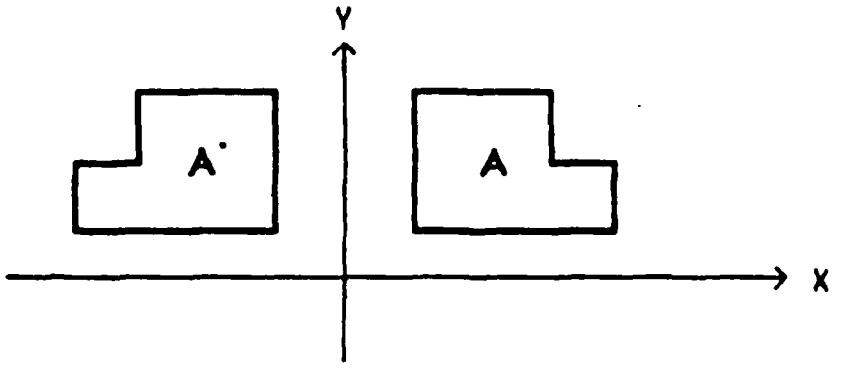


Figure 2-6: Mirror in x

(7) **my** : *mirror in y.* $A' = my(A)$ denotes that A' is the mirror image of A across the Y axis. Figure 2-7 shows the geometric meaning of this operation.

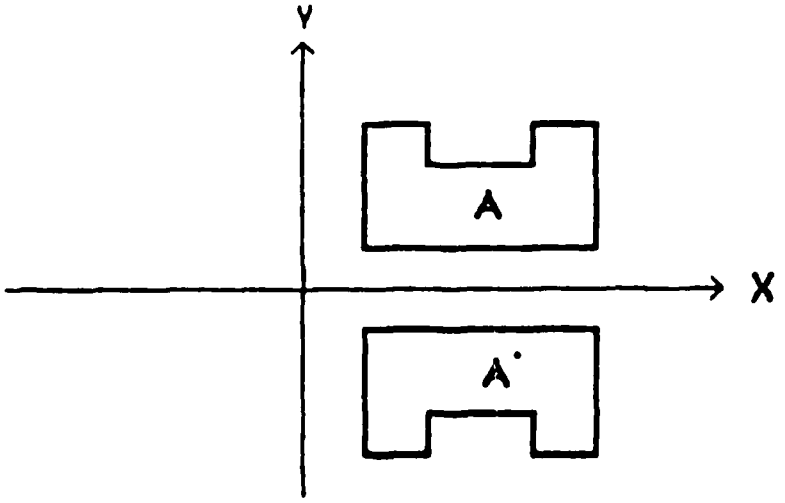


Figure 2-7: Mirror in y

These operators can be either *infix* or *prefix*. All geometric operators have the same precedence level and they are collectively left-associative in the absence of parentheses. The *interface* (which will be implemented by using **registration marks**) between two objects is not explicitly specified. This is also an application of *information hiding* and *delayed binding* on the absolute locations to make the description simple, flexible and easy to understand. Nevertheless, geometric operators do describe the relations among objects and capture the hierarchical structure of a circuit representation.

It is obvious that geometric operators are only used in describing the layout or schematic representations of a circuit when graphic interpretation is the main interest of these representations. For functional description, arithmetic operators, relational operators and logical operators are used. Of course, these operators can also be used in the description of the layout or schematic representations to represent expressions, e.g. if conditions, arguments of an arraylike structure, etc..

2.1.4.2 Other Operators

The arithmetic operators include +, -, *, /, ** (for exponentiation), and % (modulus operator). The relational operators are <=, <, == (equal to), != (not equal to), >, and >=. Each of them takes a pair of expressions as operands and returns a logical value, true or false. There are two types of logical operators. One is called *logical connectives*. They are && (AND), and || (OR). Another type is called *bitwise logical operators*. They include & (bitwise AND), | (bitwise inclusive OR), ^ (bitwise exclusive OR), and ~ (one's complement). The rules for precedence and associativity of all arithematic operators, relational operators, and logical operators follow the convention of "C"[2]. Figure 2-8 lists all the operators used in the declarative descriptions.

---

[2]Expontiation ** is the exception. It has the highest precedence.

Geometric Operators	--, |, -- ∩, |∩, rot, mx, my
Arithmetic Operators	+, -, *, /, **, %
Relational Operators	<=, <, ==, !=, >, >=
Logical Operators	&&, ||, &, |, ~, ^
Assignment Operator	=

Figure 2-8: Operators

2.1.5 Flow of Control

Two fundamental flow-of-control constructs are provided to enhance the expressiveness of a description: IF (decision making) and looping. IF is used to specify the condition. Loop is expressed in the form of (1) providing the number of times for repetition, say, ~(X(m)) means to create *m* horizontally joined instances of X; or (2) providing the upper bound, lower bound and step of the loop index, for example, |(X[i](i = 4 .. 0, -2) means to create 3 instances of X. The indices of these instances start from 4, decrement by 2 at each step, and end with 0. Thus, it creates a vertical stack with X[4] situated at the bottom, X[2] situated in the middle, and X[0] situated on the top of the stack. The EBNF definition in Appendix A depicts the syntax.

2.2 Multiple Representations

This section introduces four declarative descriptions which are used for describing a circuit. Notations introduced in the previous section will be used to illustrate the four descriptions, namely: *the layout description*, *the mixed mode description*, *the schematic description*, and *the functional description*. Each of the descriptions has its specific function in the design generation process. The layout description in conjunction with an appropriate set of leaf cells can be used to generate the layout of a circuit. The mixed mode description helps create the logic network description for the NETLIST program and simulation. The schematic description simplifies the construction of the mixed mode description. The functional description serves as an excellent reference for checking intermediate results of the circuit simulation.

A decoder provides a simple and good illustration of the efficacy of the declarative description in design and documentation. The data structure of each representation of the decoder will be shown by a tree. The correspondence between these descriptions will also be discussed. A decoder takes an n-bit number as input and uses it to select exactly one of 2^n output lines. In the VLSI generator project, **decoder** is a module generation program for a MOSIS 3 micron cmos decoder layout[3]. It currently produces decoders both in NAND (single clock) and NOR (two non-overlapping clocks) configurations. The user has to provide the design specifications in order to get the desired layout. A required specification is the number of select lines in the decoder. Other options include:

(1) the number of lambda between each decoder stage,

---

[3]The author of the decoder generator is Marty Sirkin. Readers are recommended to refer to on line VLSI Tools Manual for more details.

(2) inverting or not inverting the output,
(3) labeling the output and select lines,
(4) output file name,
(5) prefix string for the output line,
(6) prefix string for the select lines,
(7) design style (NAND or NOR),
(8) verbose mode, and
(9) printing the version number.

These options have default values. Thus, it is straightforward to generate a decoder. For example, by specifying "decoder -s 3", the layout of a 3-input-8-output NAND style decoder will be generated.

2.2.1 Layout Description

The layout description for an instance of a circuit describes how the leaf cells have to be placed to yield a specific structure and to satisfy design rules. Note that leaf cells are invariant across different instances of the same family of a circuit for a particular type of representation. In the case of the layout description, the leaf cells are created by CAESAR and CFL. They are painted pictures which carry the photo-mask information required on the fabrication process. However, what a leaf cell does internally and how it is implemented are concerns local to the leaf cell itself. The layout description only describes the relative relations among these leaf cells. The description is expressed in the form of a hierarchy, abstract at its higher levels, and progressively more detailed as it descends the hierarchy. The mechanism used is *substitution*. That is, a bigger abstract object is substituted by leaf cells or conceptually smaller abstract objects.

The following statements illustrate the layout description for a 3-to-8 NAND style decoder.

```
NAME  decoder;

TYPE  LAYOUT;
```

```
PARAMETER  n = 3;

LEAF CELLS  dec_na_ll, dec_na_i_inv, dec_na_lc, dec_na_low,
            dec_na_high, dec_na_out, dec_na_one, dec_na_zero;

FUNC  binary;

MAIN

    decoder = row [2**n] | ( | (row [i] (i = 2**n - 1 .. 0)));

    row [2**n] = dec_na_ll -- ( -- (dec_na_i_inv (n))) -- dec_na_lc;

    row [i] = dec_na_low -- select_wire [i]
            -- dec_na_high -- dec_na_out;

    select_wire [i] = ( -- (X [i,j] (j = n .. 1)));

    X [i,j] = dec_na_one, if binary (i,j) == 1
            = dec_na_zero, if binary (i,j) == 0.
```

The TYPE declaration specifies that the output of this description is a layout. A parameter n is used to indicate the number of input of the circuit. Since we want to describe a 3-to-8 decoder, n is assigned to be 3. To generate a NAND style decoder, the following leaf cells are used: **dec_na_ll, dec_na_i_inv, dec_na_lc, dec_na_low, dec_na_high, dec_na_out, dec_na_one**, and **dec_na_zero**. It is assumed that they are copied from the library into the user's working directory[4]. The internal details of **dec_na_one** and **dec_na_zero** are shown in Figures 2-9 and 2-10. Appendix B shows the rest of the leaf cells.

---

[4]The leaf cells for layout description are exactly the same as those used for the decoder generator.

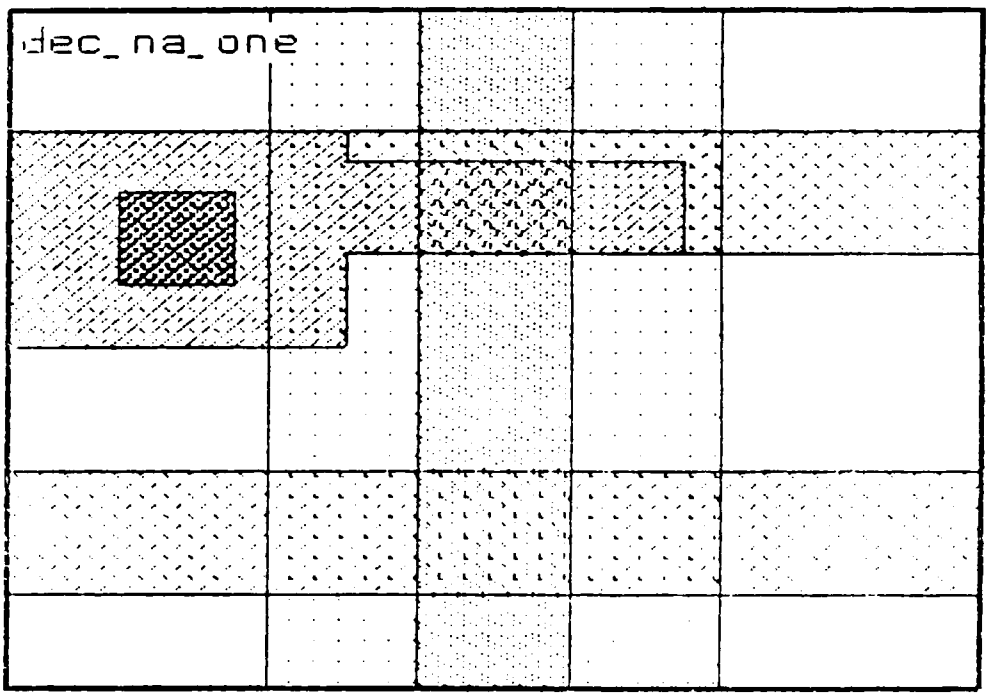


Figure 2-9: dec_na_one

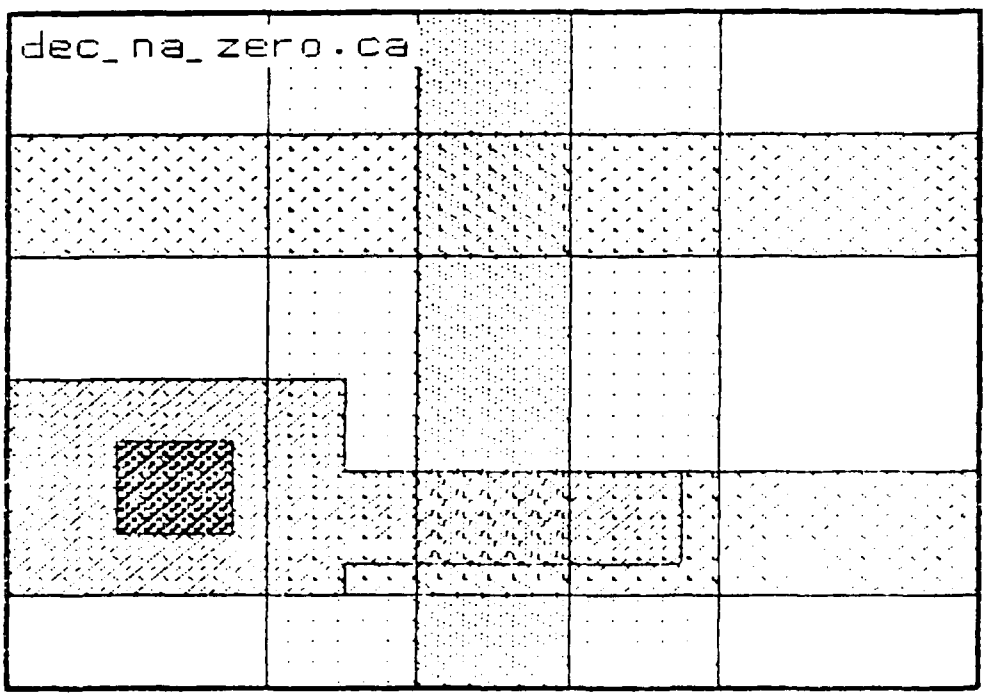


Figure 2-10: dec_na_zero

Function *binary* is imported from some other file to help in the description of the decoder. *binary* (i,j) returns 1 if the *j*th bit of the binary representation of *i* is 1 and returns 0 if it is 0.

The imperative part of the description contains information about how instances of leaf cells are to be displayed. The fragment

decoder = row [2**n] | (| (row [i] (i = 2**n - 1 .. 0)));

creates an object named **decoder** which is made up of an abstract object, row [2^n], and 2^n *vertically* stacked abstract objects named row [2^n - 1], row [2^n - 2], ..., row [0], with row [2^n - 1] situated on the top of row [2^n], and row [2^n - 2] situated on the top of row [2^n - 1], etc. At this level of abstraction, row [2^n] and row [i] can be thought of as primitive components. The network of their lower level components will be defined later. Figure 2-11 shows the geometric interpretation of this fragment. In our example, n = 3, so there are nine rows, eight for the output and one for the input.

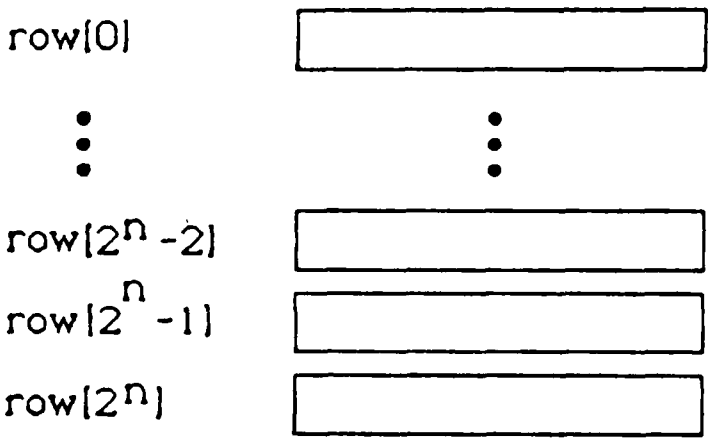


Figure 2-11: decoder = row[2**n]|(|(row[i](i=2**n-1..0)))

The fragment

```
row [2**n] = dec_na_ll -- (-- (dec_na_i_inv (n))) -- dec_na_lc;
```

specifies the components of row [2^n] one level lower in the hierarchy. Thus, row [2^n] consists of n horizontally joined instances of **dec_na_i_inv** with **dec_na_ll** on the left hand side and **dec_na_lc** on the right hand side. The geometric interpretation of this statement is shown in Figure 2-12.

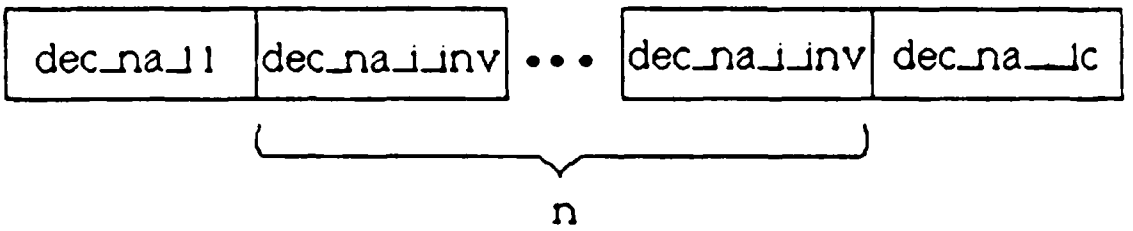


Figure 2-12: row[2**n]=dec_na_ll--(--(dec_na_i_inv(n)))--dec_na_lc

As mentioned before, the absolute locations for joins will be specified by registration marks which are information embedded in the leaf cells and are not shown explicitly in the description. Details about registration marks will be discussed in Chapter 4.

The elements in the arraylike structure **row** are further defined by the following fragment:

```
row [i] = dec_na_low -- select_wire [i]
        -- dec_na_high -- dec_na_out;
```

For i with the values from 0 to 2^n - 1, row [i] is created by horizontally joining an instance of the leaf cell **dec_na_low** and an abstract object **select_wire** [i] with **dec_na_low** on the left hand side and **select_wire** [i] on the right hand side. Then, they are placed on the left hand side of an instance of the leaf cell **dec_na_high**. Finally, this construct is horizontally joined with **dec_na_out** with the latter situated on the right hand side of the former. Figure 2-13 depicts the geometric placement.

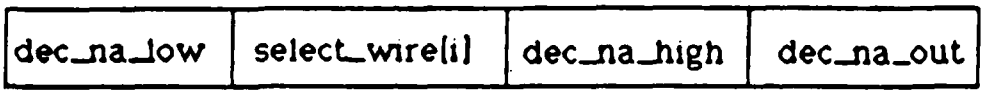


Figure 2-13: row[i]=dec_na_low--select_wire[i]--dec_na_high--dec_na_out

Moving one level down in the description hierarchy, the abstract object **select_wire** [i] is described in terms of a collection of another abstract object **X**. The fragment

select_wire [i] = (-- (X [i,j] (j = n .. 1)));

denotes that each **select_wire** consists of n horizontally joined instances of **X** with indices from n to 1, from left to right respectively. The graphic representation is shown in Figure 2-14.

Figure 2-14: select_wire[i]=(-(X[i,j](j=n..1)))

It is important to note that index i is required in the expression. It serves to distinguish different instances of **select_wire**. The definition of X clarifies this point.

Abstract object **X** is defined by the following statements:

X [i,j] = dec_na_one, if *binary* (i,j) == 1
= dec_na_zero, if *binary* (i,j) == 0;

That is, if the *j*th bit of binary representation of *i* is 1, then substitute **dec_na_one** for **X**. Otherwise, substitute **dec_na_zero** for **X**. As a result, in the case of a 3-to-8 NAND style decoder, select_wire [2] will consist of elements as shown in Figure 2-15 since the binary representation of 2 is **010**.

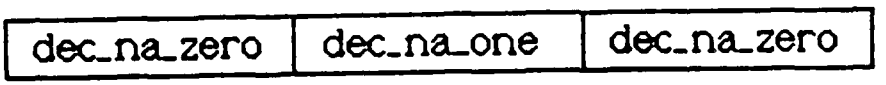


Figure 2-15: select_wire [2]

By substituting 3 for n in all the statements described, we obtain the layout representation of a 3-to-8 NAND style decoder as shown in Figure 2-16.

It should be obvious by now that many levels of abstraction are used in the layout description. Figure 2-17 shows the hierarchical structure of the objects in the layout description. The root of the tree is an instance of a decoder, while the leaves of the tree are instances of leaf cells. An arc denotes *"consists of"*. Each internal node represents an abstract object and is created by joining its children according to the relations specified in the description.

With this kind of abstraction, the complexity of the design process is reduced since the lower level objects can be thought of as primitives at each level of abstraction. The data structure that guides the generation process of decoder

Figure 2-16: Layout representation

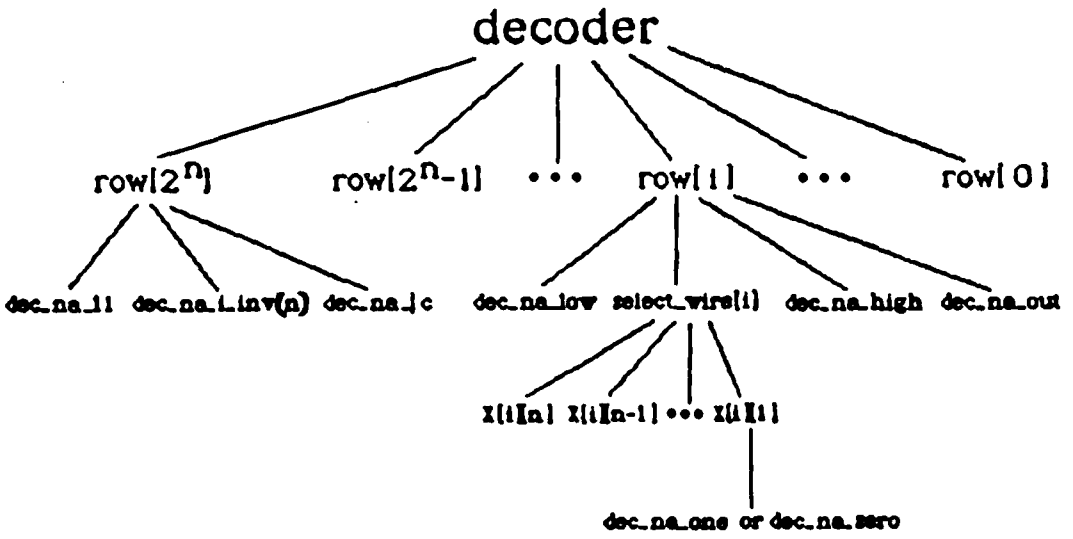


Figure 2-17: Hierarchy of objects in layout description

becomes explicit, which allows the designer to communicate his ideas and design decisions to others. Moreover, it is relatively easy to identify components of the circuit which are dependent on or independent of the parameter from the description. For example, the number of instances of **dec_na_l_inv** in row $[2^n]$ depends on the size of the input. So, for a 4-to-16 decoder, there will be 4 instances of **dec_na_l_inv**. However, the structure of row $[2^n]$ is invariant. In other words, **dec_na_ll** is always the leftmost component and **dec_na_lc** is always the rightmost component regardless of the number of instances of **dec_na_l_inv** in the middle. This observation is important in the understanding of a decoder over the entire space of parameters.

2.2.2 Mixed Mode Description

While the relations among leaf cells, which contain information about mask layers, are described in detail by its layout description, further exploitation of the power of a hierarchical description makes it necessary to provide for one higher level of abstraction -- the mixed mode description. A mixed mode description illustrates how components (gates, transistors, and intersections of wires, etc.) are connected to perform a certain function. This representation can help create the logic network description for NETLIST and simulation. A mixed mode description is also expressed in the form of a hierarchy. The interconnections of leaf cells are specified by the geometric operators. The substitution mechanism remains the same as employed in the layout description.

The mixed mode description for a 3-to-8 NAND style decoder is as follows.

```
NAME   decoder;

TYPE   MIXED;

PARAMETER   n = 3;

LEAF CELLS   gnd_mix, zero_mix, one_mix, in_mix, out_mix;

FUNC   binary;

MAIN

    decoder = row [2**n] | ( | (row [i] (i = 2**n - 1 .. 0)));

    row [2**n] = (-- (in_mix (n)));

    row [i] = -- gnd_mix -- (-- (X [i,j] (j = n .. 1))) -- out_mix;

    X [i,j] = one_mix, if binary (i,j) == 1
            = zero_mix, if binary (i,j) == 0.
```

The TYPE declaration and PARAMETER declaration indicate that the

output of this description is a mixed mode representation for a 3-to-8 decoder. The leaf cells used are **gnd_mix, zero_mix, one_mix, in_mix**, and **out_mix**. They are shown in Figure 2-18.

There is a correspondence between the leaf cells used in the mixed mode description and those used in the layout description. For example, **zero_mix** and **dec_na_zero** perform the same function. **one_mix** and **dec_na_one** are made to achieve the same designated behavior. Finally, **in_mix** is equivalent to **dec_na_i_inv**. They represent input and its complement. Similarly, the same imported function *binary* is required in the description. The return value of *binary* was discussed in section 2.2.1.

The imperative part of the mixed mode description also describes how instances of leaf cells are to be placed. In this case, a leaf cell can be a transistor, a gate, an intersection of wires or a combination of these symbols. The expansion of the output of this description is a schematic diagram of a decoder at the transistor level rather than a layout. The fragment

```
decoder = row [2**n] | ( | (row [i] (i = 2**n - 1 .. 0)));
```

will result in the same display shown in Figure 2-11 as in the corresponding statement in the layout description except for the internal details of each row. The bottom row, row [2^n], is defined as

```
row [2**n] = (-- (in_mix (n)));
```

which indicates that row [2^n] consists of n horizontally joined instances of **in_mix**. In our example, there are 3 instances of **in_mix**. The structure of the lower level components of row [2^n] is slightly simpler than the corresponding structure in the layout description. This is also true of the description of row [i]:

```
row [i] = -- gnd_mix -- (-- (X [i,j] (j = n .. 1))) -- out_mix;
```

This fragment combines the definitions of **row** [i] and **select_wire** [i] in the layout

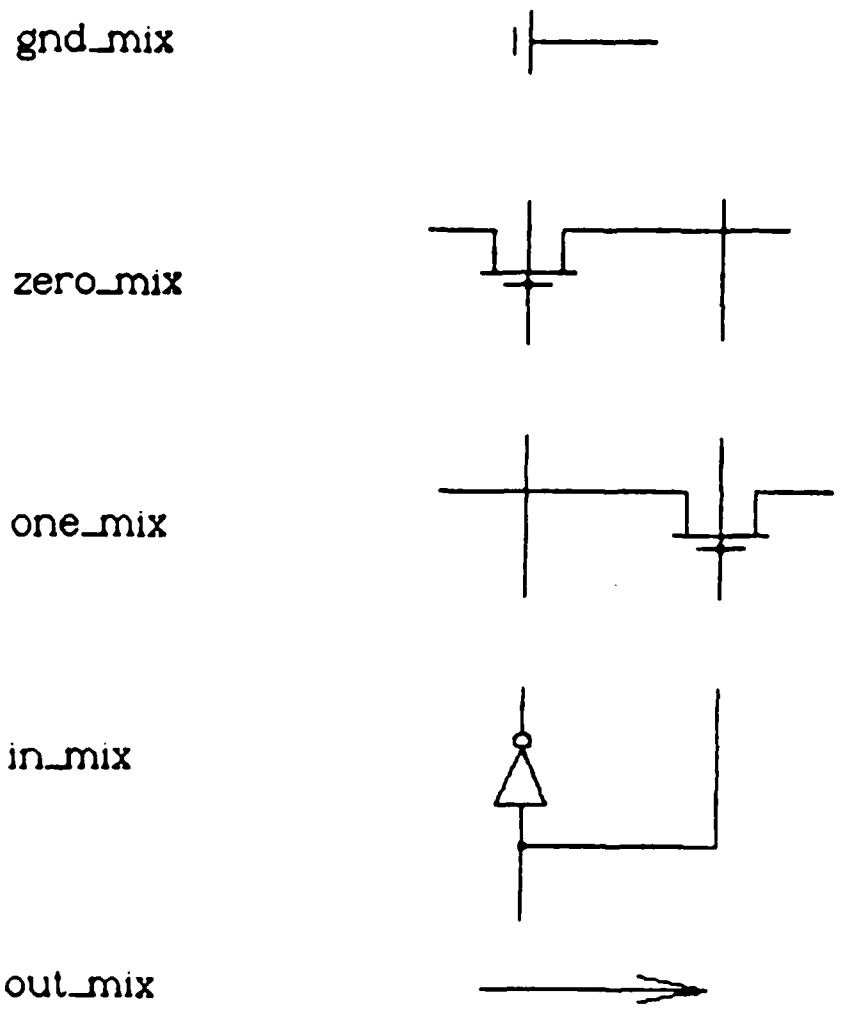


Figure 2-18: Leaf cells for mixed mode description

description. The hierarchy of objects is thus compressed. Here, **row** [i] is created by placing **gnd_mix** and out_mix on the left and right hand side of n horizontally joined abstract object **X's**, respectively. X [i,n] is the leftmost component and X[i,1] is the rightmost component. Depending on the *j*th bit of the binary representation of *i*, X [i,j] can be replaced by either **one_mix** or **zero_mix**. This is specified in the last two statements of the mixed mode description. Figure 2-19 shows the "flattened" representation of row [2] in a 3-to-8 decoder.

Figure 2-19: Flattened representation of row [2] in mixed mode description

The expansion of the output of the mixed mode description for a 3-to-8 NAND style decoder is shown in Figure 2-20.

While the abstraction mechanism for describing the decoder is the same in both the layout description and the mixed mode description, the leaf cells are different. Figure 2-21 shows the hierarchy of objects in mixed mode description. The number of levels is one less than the one in Figure 2-17. This is because the abstract object **select_wire** has been removed in the mixed mode description.

The close correspondence between these two descriptions is very important for design and documentation. The mixed mode representation provides one higher level of abstraction in circuit representation and is more immediately

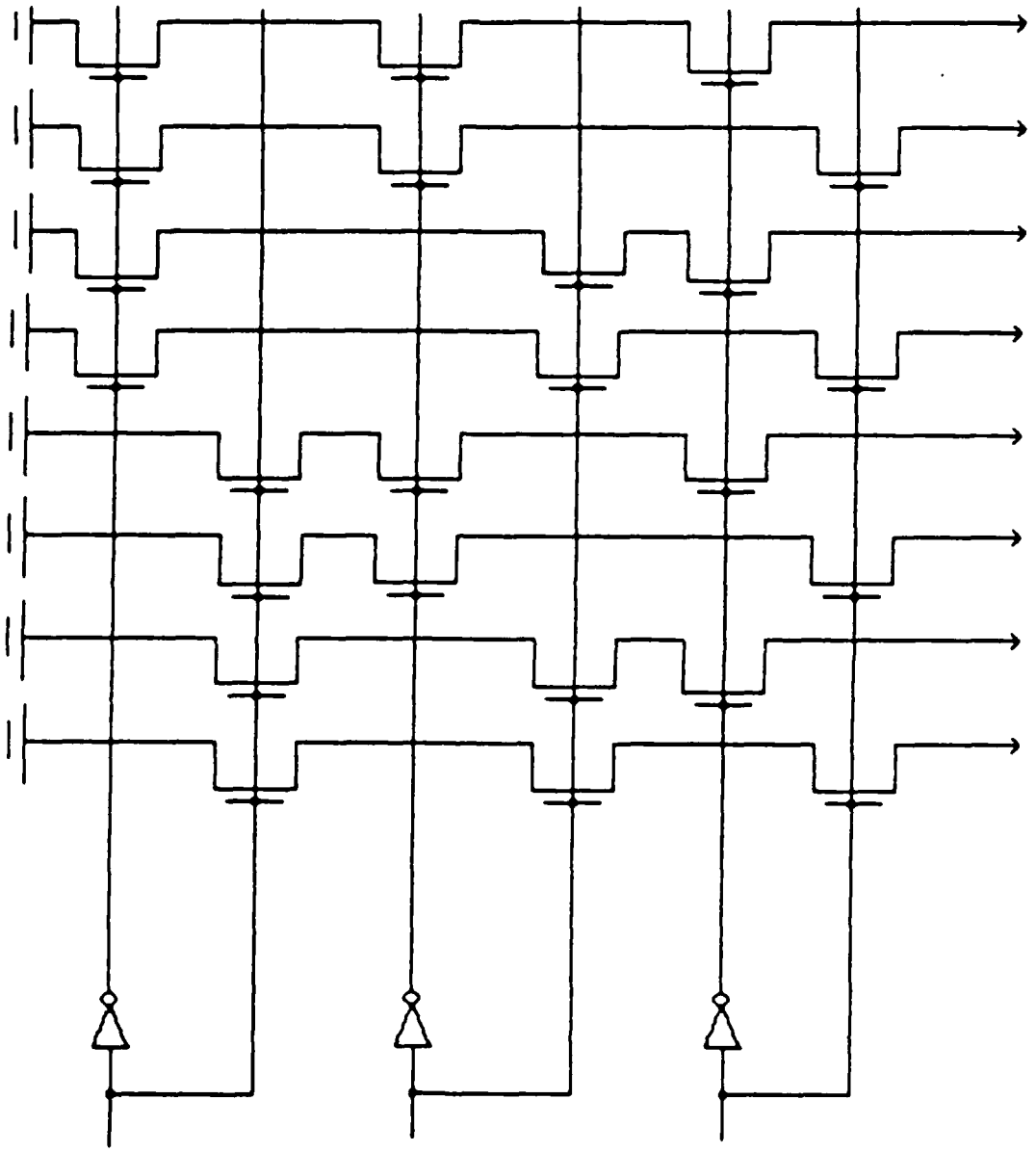


Figure 2-20: Mixed mode representation

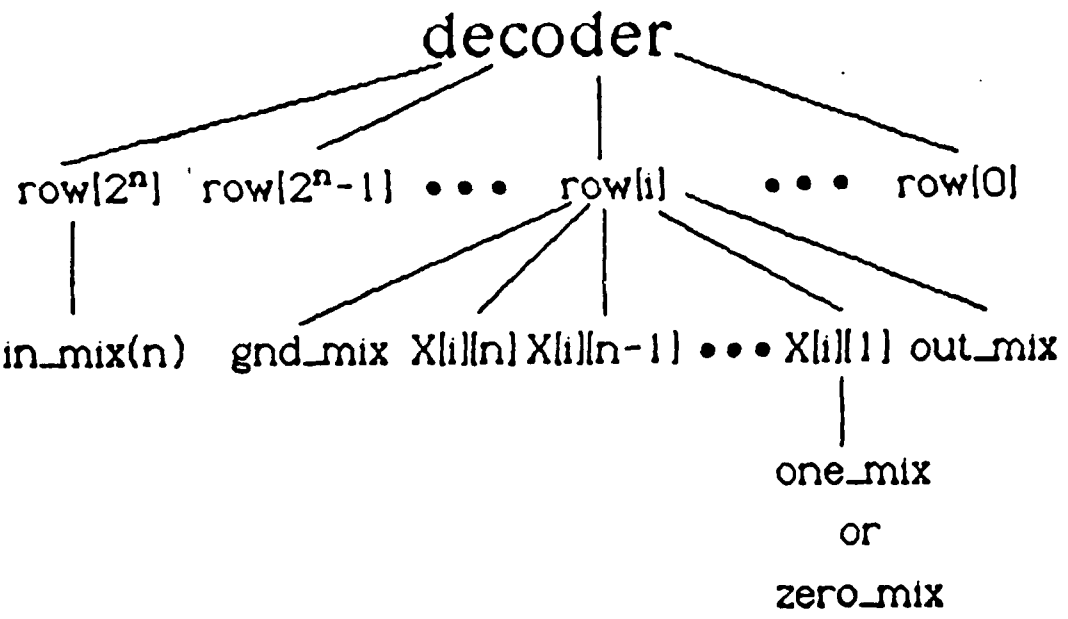


Figure 2-21: Hierarchy of objects in mixed mode description

descriptive than the layout representation. Therefore, the corresponding part in the layout representation can become more understandable to the designer and others. Moreover, suppose that after verifying the electrical correctness of the circuit through the mixed mode representation, the designer decides to change part of the design. It is relatively easy to locate the corresponding part in the layout representation and make appropriate modifications. The advantages of using multiple levels of abstraction in mixed mode description are the same as those discussed in the case of layout description.

2.2.3 Schematic Description

The schematic description provides one higher level of abstraction than the mixed mode description in the sense that the graphical version of the former is at the *gate* level, while the one in the latter is at the *transistor* level. As a result, the schematic representation is more descriptive and understandable than the mixed mode representation. The schematic description specifies the schematic diagram of a circuit with many levels of abstraction. The hierarchy of objects is intended to correspond to the hierarchies in other descriptions. Because of the correspondences between different representations, design aid and documentation are enhanced.

The following is the schematic description for a 3-to-8 NAND style decoder.

```
NAME   decoder;

TYPE   SCHEMATIC;

PARAMETER   n = 3;

LEAF CELLS   nand_sche, zero_sche, one_sche, no_connec_sche, in_sche;

FUNC   binary;

MAIN

    decoder = row [2**n] | ( | (row [i] (i = 2**n - 1 .. 0)));

    row [2**n] = (-- (in_sche (n)));

    row [i] = (-- (X [i,j] (j = n .. 1))) -- nand_sche;

    X [i,j] = (| (C [i,j,k] (k = 1 .. n)));

    C [i,j,k] = one_sche, if binary (i,j) == 1 && k == j
              = zero_sche, if binary (i,j) == 0 && k == j
              = no_connec_sche, if k != j.
```

In order to generate the schematic diagram, five leaf cells are used. They include

nand_sche, zero_sche, one_sche, no_connec_sche, and in_sche. Figure 2-22 shows the internal details of each leaf cell.

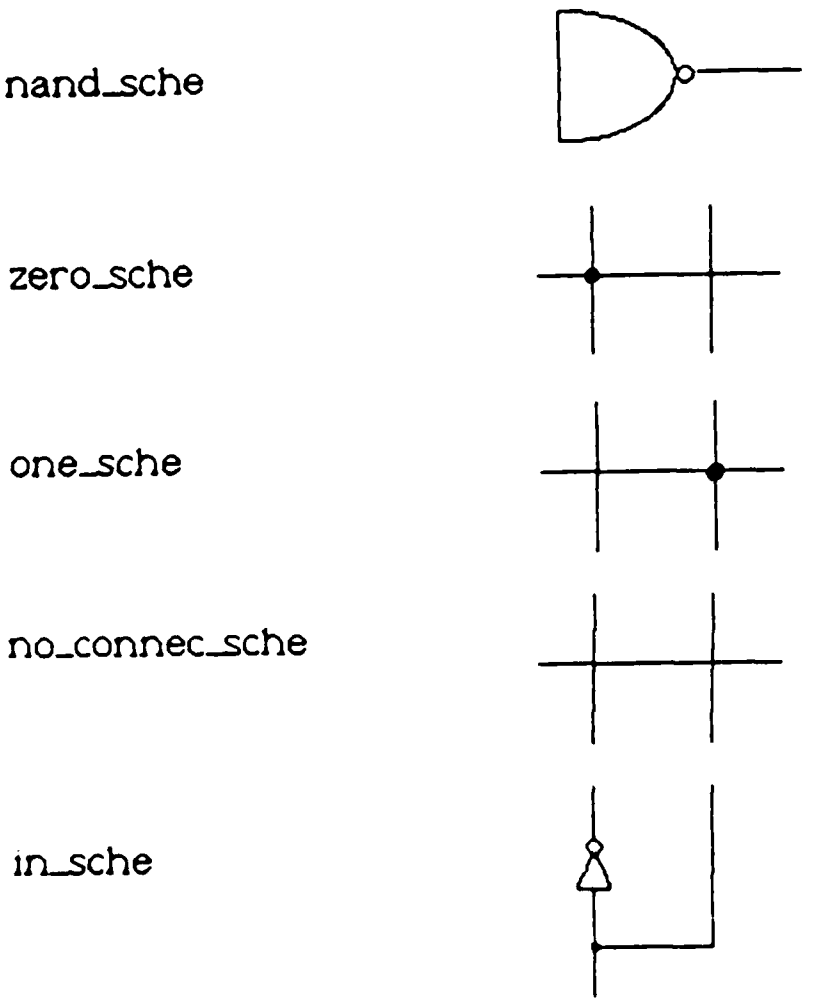


Figure 2-22: Leaf cells for schematic description

zero_sche, one_sche, and **in_sche** are equivalent to **zero_mix, one_mix,** and **in_mix** of the mixed mode description, respectively. Again, the imported function, *binary*, is used to assist in the description.

At the highest level of abstraction, decoder is defined as a network of abstract objects -- **rows**. This is also the way it is defined in the layout description and the mixed mode description. The components of row [2^n] are identical to the corresponding parts in the mixed mode description. The geometric structure of row [i] is similar to the one in the mixed mode description. Instead of replacing abstract object **X** with a leaf cell directly, the hierarchy of objects is extended one level lower. The fragment

X [i,j] = (| (C [i,j,k] (k = 1 .. n)));

means that X [i,j] consists of n *vertically* stacked objects with C [i,j,1] situated at the bottom of the stack and C [i,j,n] on the top of the stack. The leaf cells which are used in the substitution depend on the values of *k*, *j*, and the *j*th bit of the binary representation of *i*. The "flattened" representation of row [2] in a 3-to-8 decoder is shown in Figure 2-23.

zero_sche	no_connec_sche	no_connec_sche	nand_sche
no_connec_sche	one_sche	no_connec_sche	
no_connec_sche	no_connec_sche	zero_sche	

Figure 2-23: Flattened representation of row [2] in the schematic description

Figure 2-24 shows the schematic diagram of a 3-to-8 NAND style decoder.

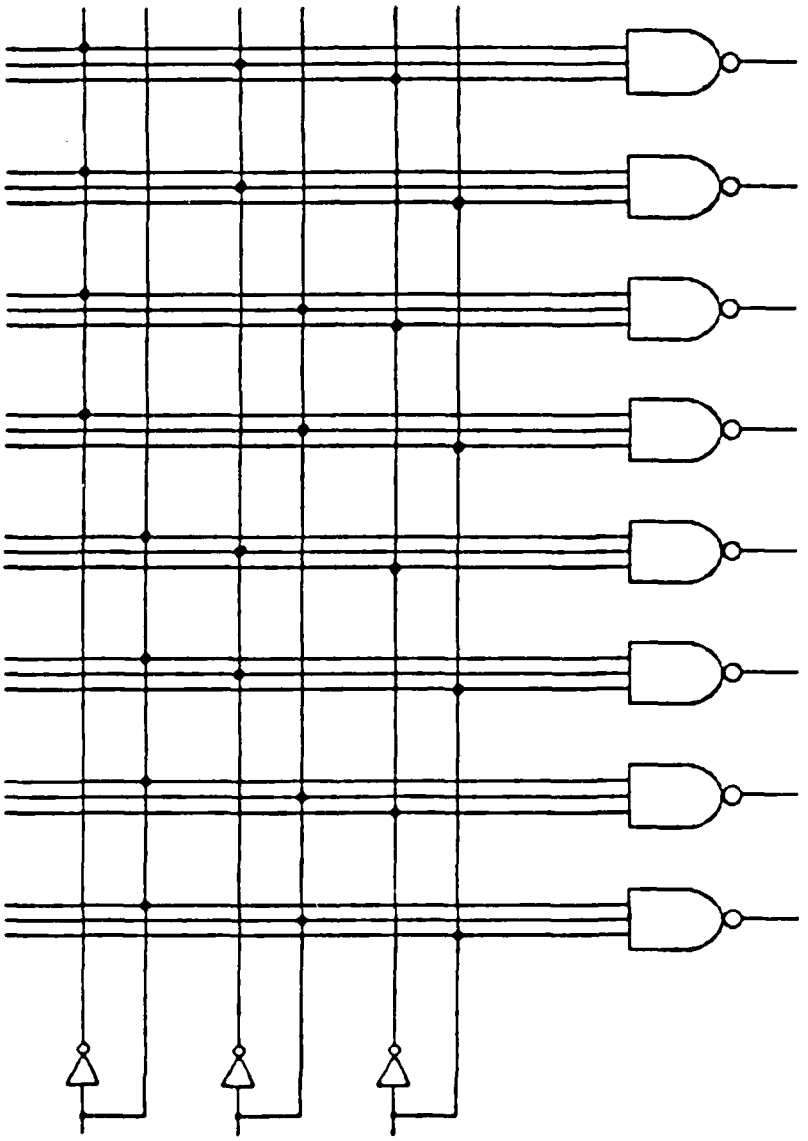


Figure 2-24: Schematic representation

The hierarchical structure of the objects used in the schematic description is shown in Figure 2-25. It is important to note that the hierarchy is extended one level lower, but the correspondence with other descriptions still holds.

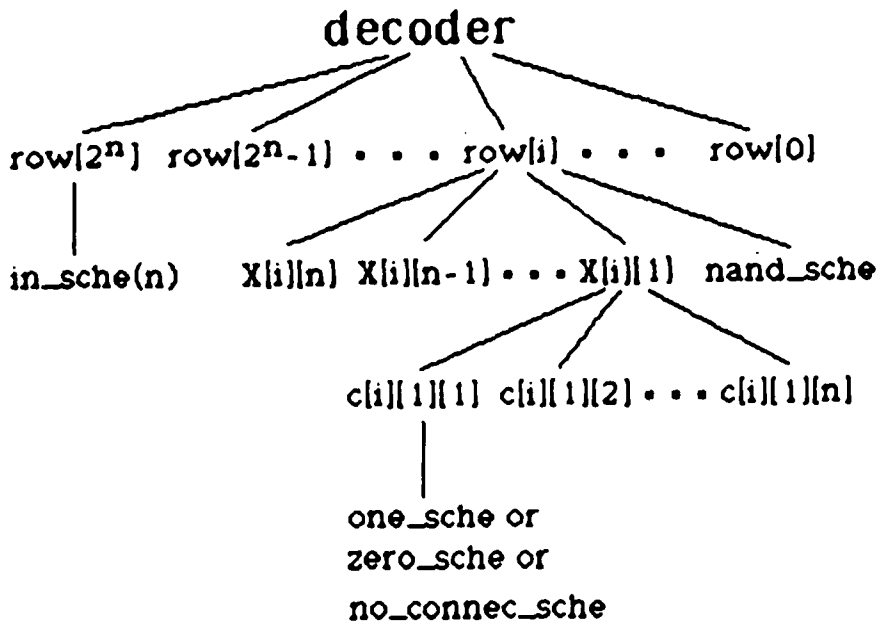


Figure 2-25: Hierarchy of objects in schematic description

2.2.4 Functional Description

The mechanisms used in the layout description, the mixed mode description, and the schematic description are the same. Each of these descriptions is expressed in the form of a **hierarchy**, and **substitution** is used to achieve many levels of abstraction. These descriptions specify the *geometric relations* of objects; therefore, they fulfill their intended purposes well when they are applied to fabricating a circuit or displaying a schematic diagram at the transistor level or the gate level. In contrast, the functional description does not specify the relative positions of objects. It describes *how* a particular design should respond to a given set of inputs. In other words, the *algorithm* to be performed by a circuit is described. The functional description serves as an excellent reference for checking the intermediate results for the circuit simulation. It provides the input and output

relationship against which the simulation results can be compared. Moreover, since a divide and conquer paradigm is used in most VLSI designs, the functional descriptions of all modules can be integrated and simulated to check the correctness of the whole design before the circuit is built. This can reduce the design cost significantly.

The following statements illustrate the functional description for a 3-to-8 NAND style decoder.

```
NAME   decoder;
TYPE   FUNCTIONAL;
PARAMETER   n = 3;
FUNC   nand, binary;
MAIN
    decoder = OUTPUT [i] (i = 0 .. 2 ** n - 1);
    INPUT = A [j] (j = n .. 1);
    OUTPUT [i] = nand (X [i,j], j = n .. 1): <timing_specification>;
    X [i,j] = binary (i,j) * A[j] + ~ binary (i,j) * ~ A[j].
```

The TYPE declaration and PARAMETER declaration indicate that this is the functional description for a 3-to-8 decoder. It should be noted that leaf cells are not used in this description since the purpose of the functional description is to describe the algorithm performed by a circuit rather than the geometric placement. Two imported functions are used in the description: *binary* and *nand*. *binary* is the same function as the one described in other descriptions. *nand* simulates the function of a NAND gate. It returns 0 if and only if all its arguments are 1.

The imperative part of the functional description specifies the internal mechanism between inputs and outputs. The statement

decoder = OUTPUT [i] (i = 0 .. 2 ** n - 1);

denotes that there are 2^n outputs. The 2^n outputs are named OUTPUT [0], OUTPUT [1], ..., and OUTPUT [2^n - 1]. For a decoder with 3 select wires, there are 8 output lines. Similarly, the fragment

INPUT = A [j] (j = n .. 1);

means that the n inputs are named A [n], A [n - 1], ..., and A [1]. Each input and output is one bit wide. Note that there is a correspondence between these two statements and the first two statements in the schematic description. OUTPUT [i] is functionally equivalent to row [i]. They describe one of the output lines. Both row [2^n] and A [j] (j = n .. 1) deal with the inputs. However, the former contains the inputs and their complements; the latter only consists of the original inputs. The complement of A [j] will be specified by the complement operator ~.

OUTPUT [i] is further defined by the function *nand*:

OUTPUT [i] = *nand* (X [i,j], j = n .. 1): <*timing_specification*>;

Since we are describing a NAND style decoder as shown in Figure 2-24, depending on the inputs, exactly one of 2^n output lines is 0; the rest are 1. In the case of a 3-to-8 decoder, suppose i = 2; then OUTPUT [2] is the result of the *nand* function of X [2,3], X [2,2], and X [2,1]. In other words, X [2,3], X [2,2], and X [2,1] are the inputs to the NAND gate and OUTPUT [2] is asserted (= 0) if and only if X[2,3] = X[2,2] = X[2,1] = 1. The <*timing_specification*> specifies when the output bit becomes stable and available for external circuit. It can be expressed in terms of i, clock period (T), and time delay (td). The values of T and td are technology and implementation dependent.

Depending on the *j*th bit of the binary representation of *i*, i.e. *binary* (i,j), the value of X [i,j] is either the input A [j] or its complement $\overline{A[j]}$. Note that given an OUTPUT [i], the binary representation of i corresponds to the inputs A[n] A[n-1]

... A[j] ... A[1], where A [n] * 2^{n-1} + A [n -1] * 2^{n-2} + ... + A [1] * 2^0 = *i*. Note also that for an asserted OUTPUT [i], the inputs to the NAND gate must be all 1's. The algorithm is thus the following:

X [i,j] = A [j], if *binary* (i,j) == 1;
X [i,j] = $\overline{\text{A [j]}}$, if *binary* (i,j) == 0;

For example, OUTPUT [5] is asserted only when the inputs are 101. That is, A [3] = 1, A [2] = 0, and A [1] = 1. Therefore, if the input is 101, according to the algorithm, X [5,3] = A [3] = 1, X [5,2] = $\overline{\text{A [2]}}$ = 1, and X [5,1] = A [1] = 1. OUTPUT [5] becomes 0 which is asserted. In contrast, if the input is 110, then X[5,3] = A[3] = 1, X[5,2] = $\overline{\text{A[2]}}$ = 0, and X[5,1] = A[1] = 0. The result of the *nand* function with these three arguments is 1. Since *binary* (i,j) is either 1 or 0, the algorithm can be simplified by stating that

X [i,j] = *binary* (i,j) * A [j] + ~ *binary* (i,j) * ~ A [j].

While the functional description uses arithmetic operators rather than geometric operators to describe a circuit, it still corresponds with the other descriptions. The mechanism which it uses is substitution and the structure of the description is hierarchical in nature. Figure 2-26 shows the hierarchy in the functional description.

The hierarchy is similar to those in the layout description, the mixed mode description, and the schematic description. The major difference between them is that the leaves of the tree in the functional description are inputs or their complements while the leaves of the trees in other descriptions are leaf cells. Because of the correspondence, the functional description serves as an excellent reference in simulation and a good documentation of the algorithm performed by a circuit.

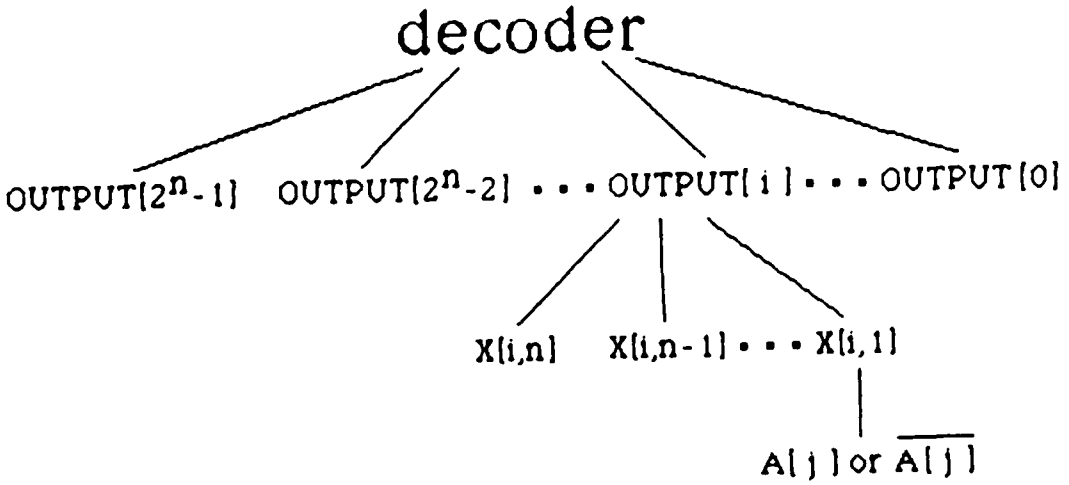


Figure 2-26: Hierarchy in functional description

2.3 A More Complex Example -- Multiplier

Section 2.2 showed the reader how to use the set of notations introduced in Section 2.1 to describe the multiple equivalent representations of a decoder. This section will present a more complex example, a **multiplier**, to illustrate the versatility of the notations. **mult** is a generator for constructing an M x N cmos multiplier layout[5]. The latest version (Jan. 1986) of the **mult** generator provides the user with the following options: (1) set the number of bits in the multiplicand operand, (2) set the number of bits in the multiplier operand, (3) define the left side horizontal bus as ground or vdd, (4) label the product output bits, (5) make the number representation signed (two's complement) or unsigned, (6) label the multiplicand input bits, (7) label the multiplier input bits, (8) use one more adder

[5]The author of the multiplier generator is Wayne Winder.

to facilitate accumulation, (9) turn the internal cell labels on or off, (10) specify the width of horizontal/vertical GND/VDD bus and multiplicand/ multiplier gate sizes, and (11) debug. Each of these options has a default. For example, to create a 3 x 4 cmos multiplier layout with 2's complement number representation, left side bus being GND, and other options being defaults, the user can simply specify "**mult -m 3 -n 4**". The procedure is as simple as the one used in generating a decoder.

A 3 x 3 signed two's complement multiplier is chosen as an example in our discussion. In order to give the reader a better understanding of the different descriptions for the 3 x 3 multiplier, the algorithm used by the multiplier generator in creating the product output is described in section 2.3.1. Section 2.3.2 gives the schematic description and the functional description.

2.3.1 Algorithm

The multiplication process may be viewed as having two parts: (1) the generation of partial products, and (2) the reduction of these partial products into a final product output. This section reviews the important details of the algorithm used by Winder in his design of the *mult* generator [Winder 84].

2.3.1.1 Unsigned Multiplication

The most basic form of multiplication consists of forming the product of two *unsigned* binary numbers. Let the two inputs to the multiplier be defined as the multiplicand (**X**) with **m** bits, and the multiplier (**Y**) with **n** bits. The product of X*Y is defined as **P** (**P** has n+m bits). The process of multiplication is illustrated in Figure 2-27.

Each partial product (e.g. $X_{m-1}Y_0 \ldots X_2Y_0\ X_1Y_0\ X_0Y_0$) is conditional on the multiplicand and one of the multiplier bits. In other words, the evaluation of partial products consists of the logical ANDing of the multiplicand and the relevant

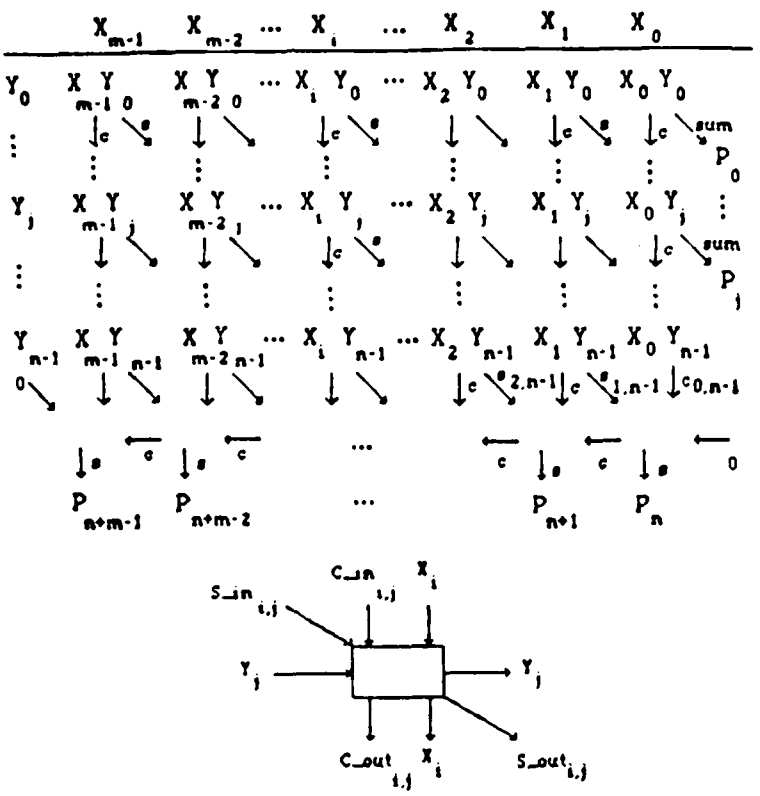


Figure 2-27: Multiplication of two unsigned binary numbers

multiplier bit. The successive additions and shifts for $0 \le i \le m-1$ and $0 \le j \le n-1$ are accomplished as follows. First the $sum\_in_{i,j}$, $carry\_in_{i,j}$ and X_iY_j are added together. Then, $sum\_out_{i,j}$ and $carry\_out_{i,j}$ are sent down to the next row. The $sum\_out_{i,j}$ becomes $sum\_in_{i-1,j+1}$ and $carry\_out_{i,j}$ becomes $carry\_in_{i,j+1}$. Note that the carry overs are not rippled through the next higher order bits. The sum_out's and carry_out's generated by the addition with the current partial products are added in with the next partial product. Note also that elements in the top row and the leftmost column in Figure 2-27 are considered as having 0's as sum_in and carry_in. The sum_out's of elements in the rightmost column become the lower order n bits of the final product. The n+1 st row is implemented by a ripple adder

where all the sum_out's become the higher order bits of the final product after the ripple addition. For example, P_n is the sum_out of adding $carry\_out_{0,n-1}$ and $sum\_out_{1,n-1}$; P_{n+1} is the sum_out of adding $carry\_out_{1,n-1}$, $sum\_out_{2,n-1}$, and the carry_out from the addition performed for P_n.

2.3.1.2 Signed Two's Complement Multiplication

The algorithm used in the case of *signed* multiplication is more complex since the most significant bit of each number is the sign bit. Let X, the multiplicand, and Y, the multiplier, be represented by $(X_{m-1}X_{m-2},\ldots,X_2X_1X_0)$ and $(Y_{n-1}Y_{n-2},\ldots,Y_2Y_1Y_0)$. For positive values of X and Y, the most significant bits, X_{m-1} and Y_{n-1}, are zero. In this case, the multiplication process can be accomplished with the algorithm for unsigned multiplication as described in section 2.3.1.1. However, when the sign bits are not zero, we must revisit the significance of the two's complement representation. For example, -5 is represented as 1011 in the 2's complement representation (assuming a 4-bit word). The two's complement process is used to eliminate the necessity of using the sign notation by embedding the negative value of the number in the most significant bit. Thus 1011 is made up of two parts: the negative part $(-1)2^3 = -8$ and the positive part, 011, which is 3. Adding these two, we see the significance of $-8 + 3 = -5 = 1011$.

Hence, arithmetic using two's complement is predicated on the assumption that the two operands have the same number of bits. If we add two two's complement numbers without aligning the sign bits, the result would be incorrect as illustrated below.

$$\begin{array}{rcl} 1011 & = & -5 \\ 10111 & = & -9 \\ \hline 100010 & = & -30 \end{array}$$

Alignment of the sign bit of a negative number to a higher order number can be

simply accomplished by filling with leading 1's to the negative number with less bits until the two numbers have the same number of bits. This is because $-2^{m-1} = -2^m + 2^{m-1}$. For instance, -5 = 1011 = 11011 = 111011 = ... This idea is used for aligning the sign bit of the **negative partial product** to the sign bit of the next higher order partial product if they are not on the same bit position. For a non-negative number, only leading zeros need be added.

Given two numbers X and Y in the 2's complement representation where the multiplicand X is of the form $(X_{m-1}\ X_{m-2}, \ldots, X_2\ X_1\ X_0)$ and the multiplier Y is of the form $(Y_{n-1}\ Y_{n-2}, \ldots, Y_2\ Y_1\ Y_0)$, we can write

$$X = -X_{m-1}2^{m-1} + X_{m-2}2^{m-2} + \ldots + X_1 2^1 + X_0 2^0,$$

$$Y = -Y_{n-1}2^{n-1} + Y_{n-2}2^{n-2} + \ldots + Y_1 2^1 + Y_0 2^0,$$

and

$$P = X * Y$$
$$= -P_{m+n-1}2^{m+n-1} + P_{m+n-2}2^{m+n-2} + \ldots + P_1 2^1 + P_0 2^0$$

where P_k is expressed in terms of X_i and Y_j. To obtain the value of P_k, we first multiply X by Y_0 to get the first partial product:

$$-Y_0X_{m-1}2^{m-1} + Y_0X_{m-2}2^{m-2} + \ldots + Y_0X_1 2^1 + Y_0X_0 2^0$$

As mentioned before, $-2^{m-1} = -2^m + 2^{m-1}$. The first partial product is equal to

$$-Y_0X_{m-1}2^m + Y_0X_{m-1}2^{m-1} + Y_0X_{m-2}2^{m-2} + \ldots + Y_0X_1 2^1 + Y_0X_0 2^0.$$

In this manner, the sign bit is shifted one bit to the left, thus accomplishing the alignment of the sign bit with the one in the second partial product. The second partial product is

$$-Y_1X_{m-1}2^m + Y_1X_{m-2}2^{m-1} + \ldots + Y_1X_1 2^2 + Y_1X_0 2^1$$

When it is added to the first partial product, the sum becomes

$$-(Y_0X_{m-1} + Y_1X_{m-1})2^m + (Y_0X_{m-1} + Y_1X_{m-2})2^{m-1} + \ldots + Y_0X_0 2^0.$$

By the same token, the sign bit should be aligned with the sign bit of the next partial product. Let us now formally define the values of the sign extension bits by rewriting the above cumulative sum in the following way:

$$-f_1 2^{m+1} + g_1 2^m + (Y_0X_{m-1} + Y_1X_{m-2})2^{m-1} + \ldots + Y_0X_0 2^0$$

Consider the table in Figure 2-28.

Y_0 X_{m-1}	Y_1 X_{m-1}	f_1	g_1
0	0	0	0
0	1	1	1
1	0	1	1
1	1	1	0

Figure 2-28: Sign extension bit

If both Y_0X_{m-1} and Y_1X_{m-1} are non-negative, we fill the extended bit (f_1) with 0 which corresponds to the fact that the sum of two non-negative numbers is non-negative. If one is positive and the other is negative, the result should be negative (since the product will be negative). By applying the observation that $-2^m = -2^{m+1} + 2^m$, we derive $f_1=g_1=1$ and the alignment is thus accomplished. When both numbers are negative, adding these two sign bits results in overflow. In this case, the sign bit (i.e. overflow bit) still represents a negative number and no alignment is needed since $-2^m + -2^m = -2^{m+1}$.

From table 2-28, it is obvious that f_1 = OR (Y_0X_{m-1},Y_1X_{m-1}) and g_1 = XOR (Y_0X_{m-1},Y_1X_{m-1}). Since $f_0 = Y_0X_{m-1}$ and $g_0 = Y_0X_{m-1}$, we derive f_1 = OR

$(f_0, Y_1 X_{m-1})$ and $g_1 = \text{XOR}\ (f_0, Y_1 X_{m-1})$. Similarly, subsequent sign extension bits f_j and g_j (for $j < n - 1$) will have the values

$$f_j = \text{OR}\ (f_{j-1}, Y_j X_{m-1})$$

and

$$g_j = \text{XOR}\ (f_{j-1}, Y_j X_{m-1}).$$

This process of forming the partial product, adding it with the previous cumulative sum and performing the sign extension is carried out starting from Y_0 until Y_{n-2}. At each step, one corresponding bit of the final result is produced, i.e. $P_0, P_1, \ldots, P_{n-2}$. The final partial product involving Y_{n-1} has to be treated differently: (1) If $Y_{n-1} = 0$, the sign extension process in the preceding P_k has taken care of the correction needed if $X_{m-1} = 1$; (2) If $Y_{n-1} = 1$, it can be shown that we need to add $2^{m+n-1} - X$ to the final sum in order to get the correct result. This can be achieved by two's complementing X before adding it to the cumulative sum. Since this 2's complement will not change anything if $Y_{n-1} = 0$, the procedure is the same independent of the value of Y_{n-1}. That is, the last partial product is the same as

$$(X_{m-1} Y_{n-1} - Y_{n-1}) 2^{m+n-2} + Y_{n-1}(1 - X_{m-2}) 2^{m+n-3} + \ldots +$$
$$Y_{n-1}(1 - X_0) 2^{n-1} + Y_{n-1} 2^{n-1}$$

Observe that $(1 - X_i) = \overline{X_i}$. The sign extension process for the last cumulative sum has the modified formula

$$f_{n-1} = g_{n-1} = \text{OR}\ (f_{n-2}, \text{AND}\ (\overline{X_{m-1}}, Y_{n-1})).$$

The block diagram of the two's complement implementation is shown in Figure 2-29. It should be noted that after the (n-1)th addition, the result, carry_outs, sum_outs and the Y_{n-1} bit, have overlapping binary weight values. For instance,

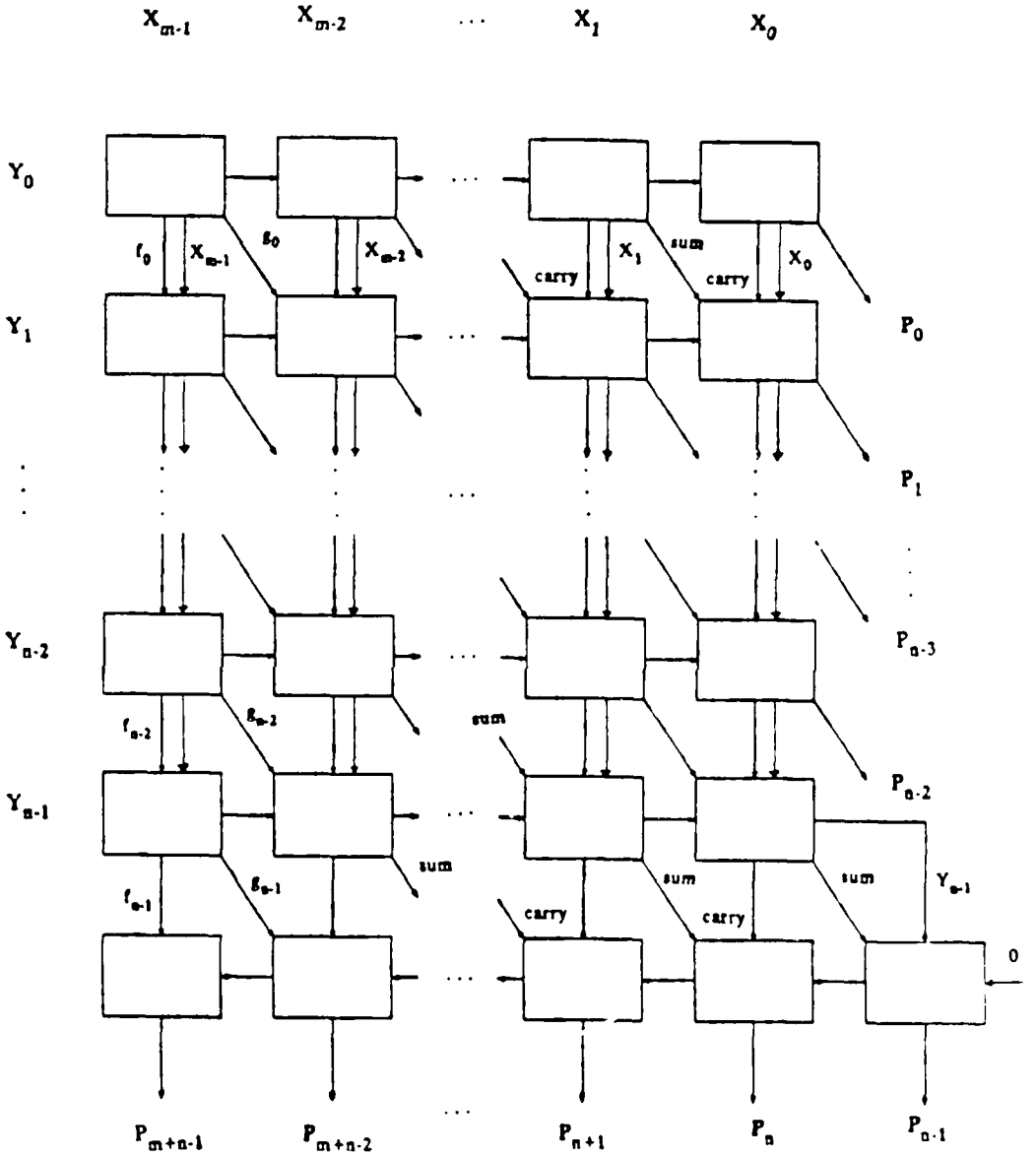


Figure 2-29: Block diagram of the two's complement implementation

the carry out from the *i*th column has the same weight as the sum out from the (*i*+1)th column. The final stage of the multiplier is thus implemented by a ripple adder to obtain the higher order *m*+1 bits of the final product. These bits, in conjunction with the P_{n-2} P_{n-3}, ..., P_0 obtained in the first n-1 partial product additions, constitute the two's complement representation of the final product.

2.3.2 Descriptions

The layout description, the mixed mode description, the schematic description and the functional description of the multiplier apply the same mechanisms as those which are used for describing the decoder. Each description is a *hierarchical* structure in which the building blocks are objects except for the functional description. Objects may correspond to leaf cells, or they may be abstract objects which are defined in terms of lower level objects, which may again be defined in terms of even lower level objects, etc. By using the *substitution* mechanism, the complex geometrical patterns of the layout and schematic diagram for a multiplier can be described in a clean and understandable manner. This section shows the schematic description and the functional description of a 3 x 3 signed two's complement multiplier. The layout description and the mixed mode description are given in Appendices C and D, respectively.

2.3.2.1 The schematic Description

As pointed out in Section 2.2.3, the graphical version of the schematic description is at the gate level. Thus the schematic description provides a higher level of abstraction than the layout description or the mixed mode description. The following statements define the schematic description for a 3 x 3 signed two's complement multiplier.

```
NANE   multiplier;
```

```
TYPE  SCHEMATIC;
PARAMETER  m = 3, n = 3;
LEAF CELLS  SignExt, FullMult, LSignExt, Comp, RComp, Add;
MAIN
    multiplier = adder | row[n] | (| (row[i] (i = n - 1 .. 1)));
    row[i] = SignExt -- (-- (FullMult (m - 1)));
    row[n] = LSignExt -- (-- (Comp (m - 2))) -- RComp;
    adder = (-- (Add (m + 1))).
```

To generate a signed two's complement multiplier, six leaf cells are needed. Figures 2-30, 2-31, 2-32, 2-33, 2-34, and 2-35 show the block diagram and internal gate representation for each leaf cell.

SignExt generates the sign extension bits f_j and g_j for $0 \le j \le n\text{-}2$, while **LSignExt** evaluates the last sign extension bits, f_{n-1} and g_{n-1}. The function of **FullMult** is to calculate the sum of $carry\_out_{i,j}$, $sum\_out_{i+1,j}$ and the ANDed function of X_i and Y_j for $0 \le i \le m\text{-}2$ and $0 \le j \le n\text{-}2$. **Comp** and **RComp** evaluate the sum of $carry\_out_{i,n-2}$, $sum\_out_{i+1,n-2}$ and the ANDed function of $\overline{X_i}$ and Y_{n-1}, $0 \le i \le m\text{-}2$. They are basically the same with the exception that Y_{n-1} in **RComp** exits from the right hand side and curves down to the first bit of the ripple adder. Y_{n-1} is one of the inputs to the lowest bit of the ripple adder because of the last term in the expanded version of the nth partial product, $Y_{n-1}2^{n-1}$. The leaf cell **Add** computes the sum of three inputs and produces one bit of the final product. In addition, the carry out is sent to the next higher order bit position of the ripple adder.

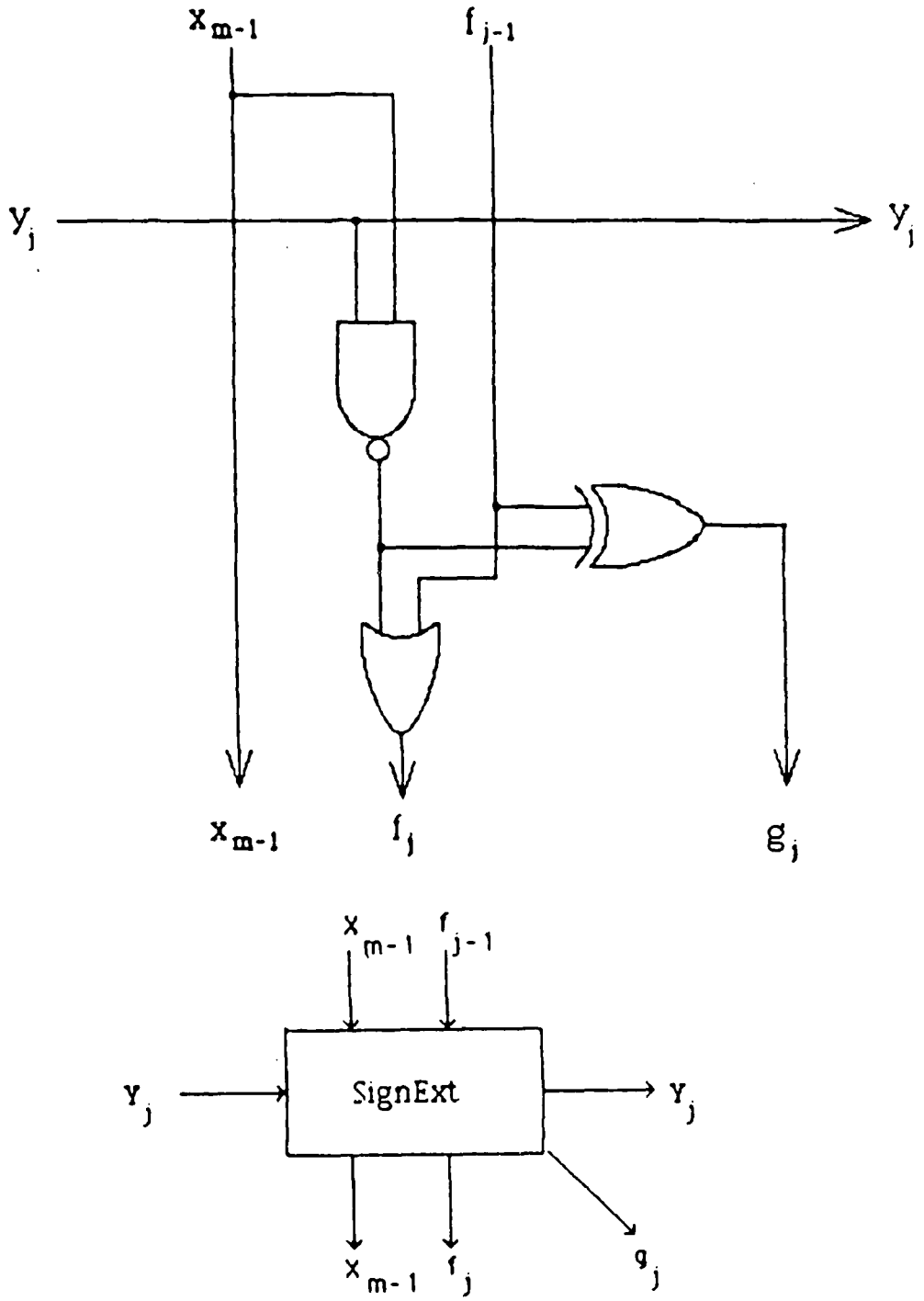


Figure 2-30: SignExt

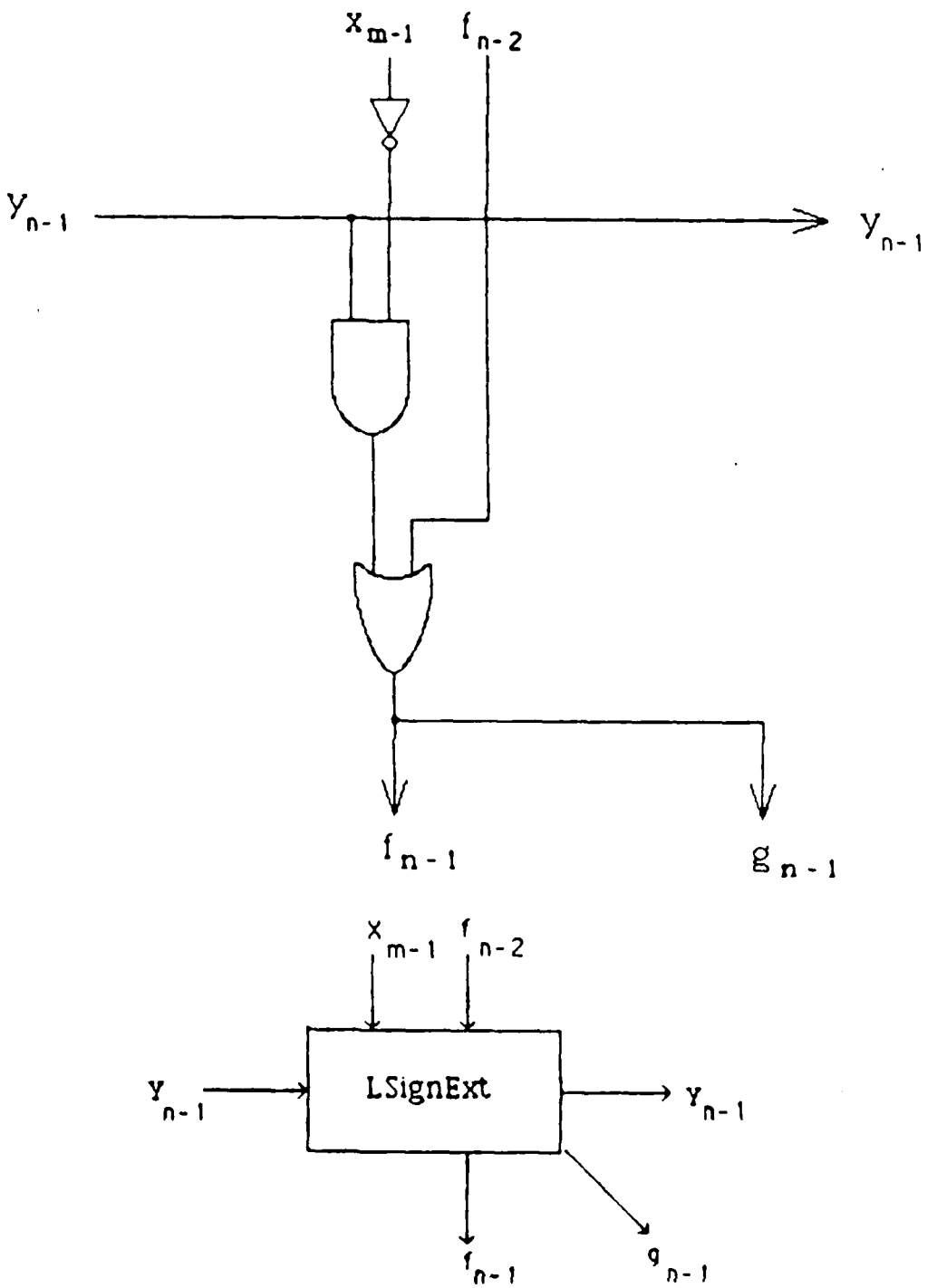


Figure 2-31: LSignExt

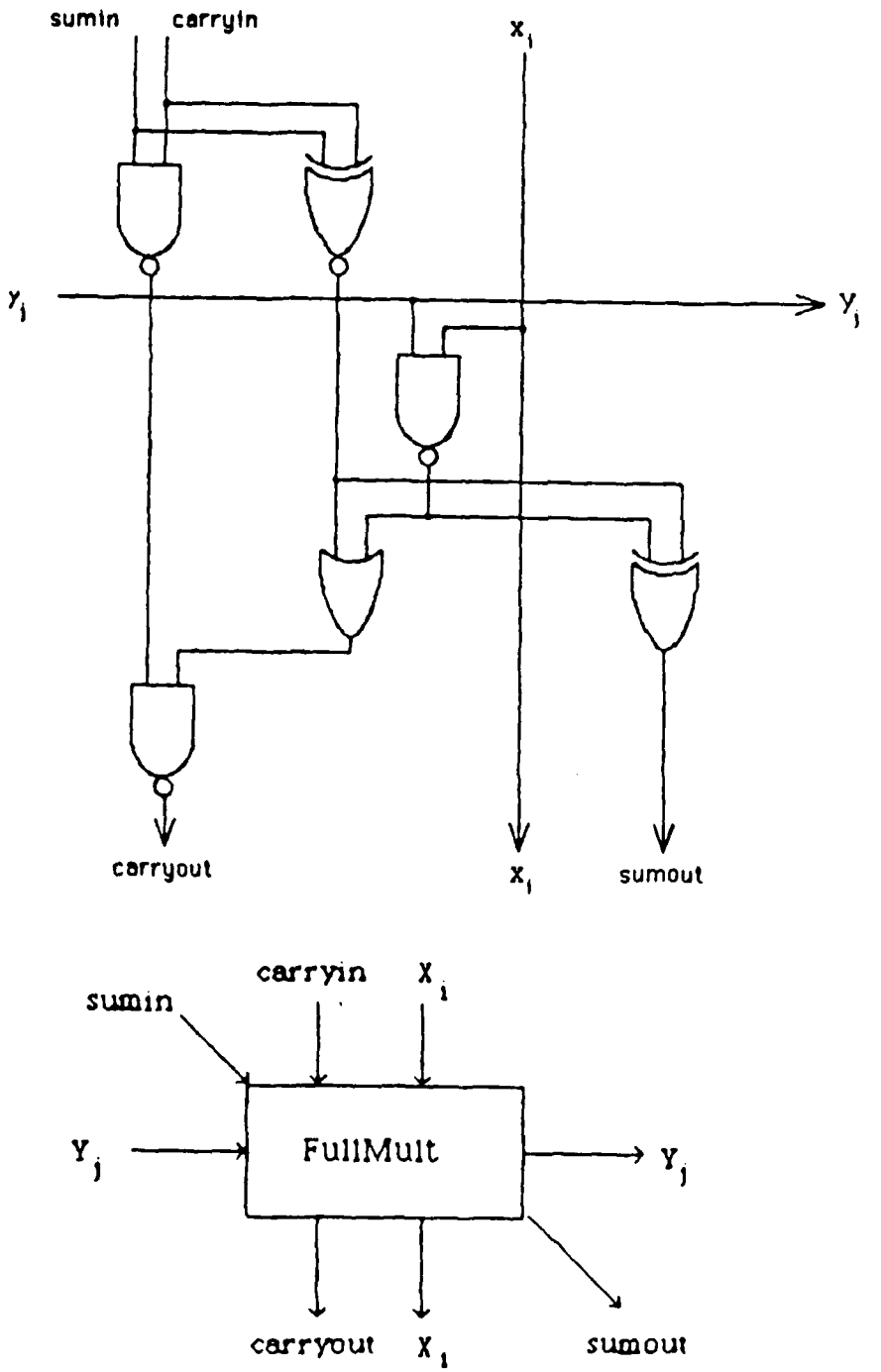


Figure 2-32: FullMult

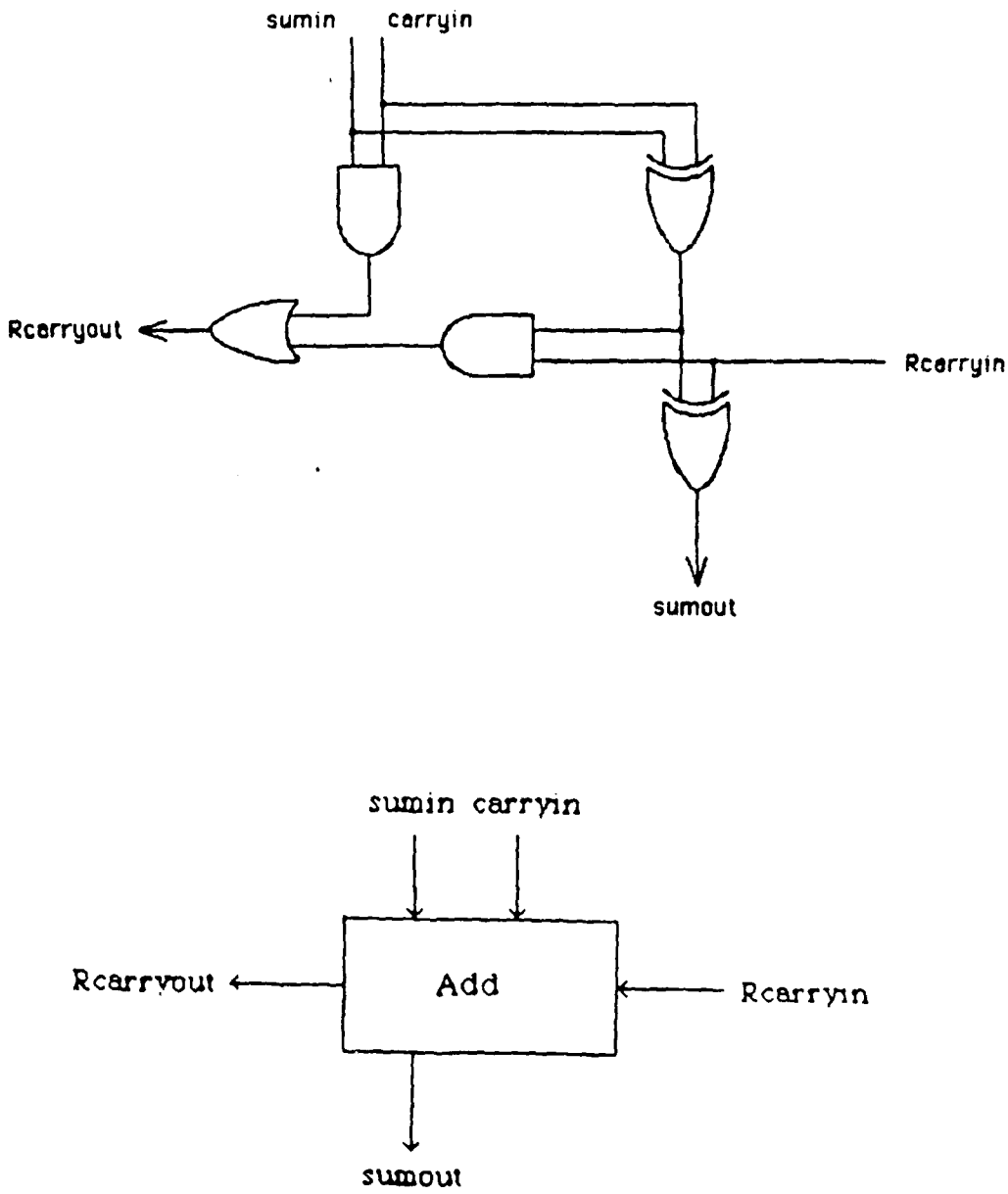


Figure 2-33: Add

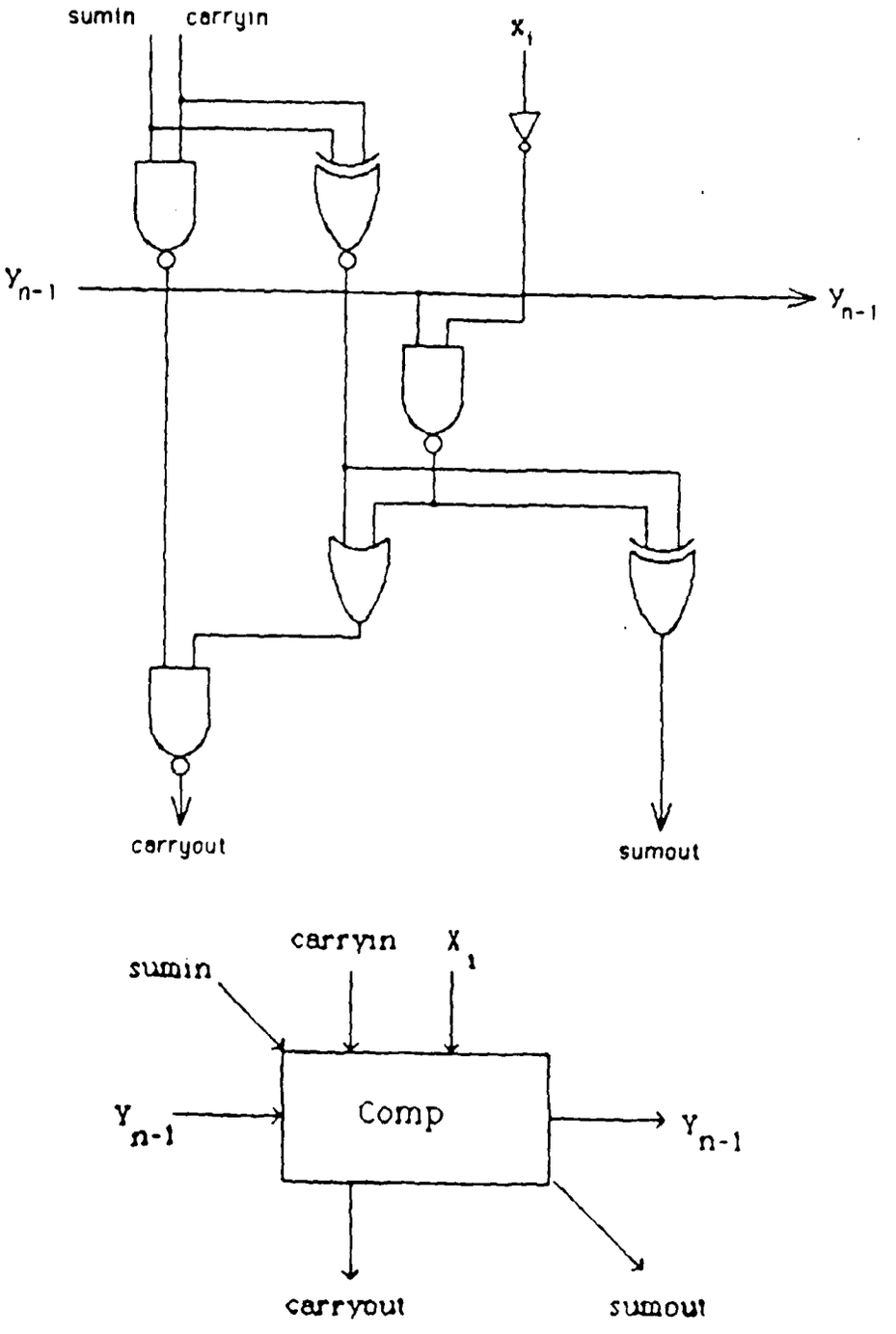


Figure 2-34: Comp

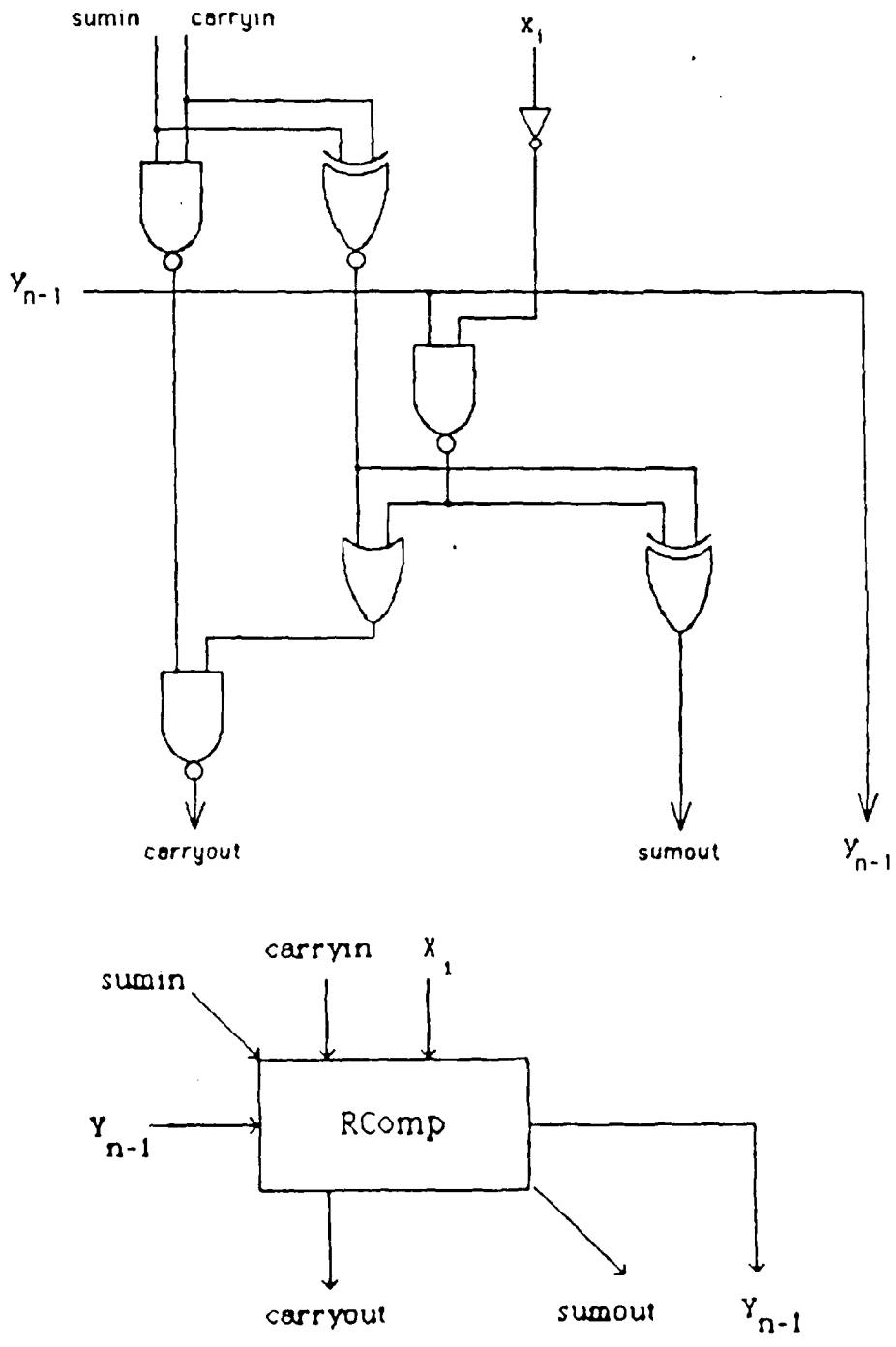


Figure 2-35: RComp

At the highest level of abstraction, multiplier is defined as

multiplier = adder | row[n] | (| (row[i] (i = n - 1 .. 1)));

Thus it consists of an adder and n rows named row[n], row[n-1], ..., row[1]. These objects are vertically stacked. The adder is situated at the bottom of the stack while row[1] is situated on the top of the stack. For a 3 x 3 multiplier, there are three rows. The statement

row[i] = SignExt -- (-- (FullMult (m - 1)));

denotes that row[i] is made up of SignExt horizontally joined with m-1 FullMult's. SignExt is situated on the left hand side and the FullMult's are situated on the right hand side. The m-1 FullMult's are also horizontally joined. From the previous descriptions of SignExt and FullMult, it is obvious that row[i] accomplishes the process of forming the partial product, adding it with the previous cumulative sum and performing the sign extension. The statement

row[n] = LSignExt -- (-- (Comp (m - 2))) -- RComp;

specifies that row[n] consists of m-2 horizontally joined instances of Comp with LSignExt on the left-end and RComp on the right-end. Note that the function of row[n] is to perform the ANDing of the complement of multiplicand and Y_{n-1} as well as generate the cumulative sum and the last pair of sign extension bits, f_{n-1} and g_{n-1}. The elements in the adder are further defined by the following fragment

adder = (-- (Add (m+1)));

Therefore in a 3 x 3 multiplier there are 4 Add cells horizontally joined. The object adder generates the higher order $m+1$ bits of the final product. Figure 2-36 shows the expansion of the schematic description for a 3 x 3 signed two's complement multiplier which is an instance of Figure 2-29.

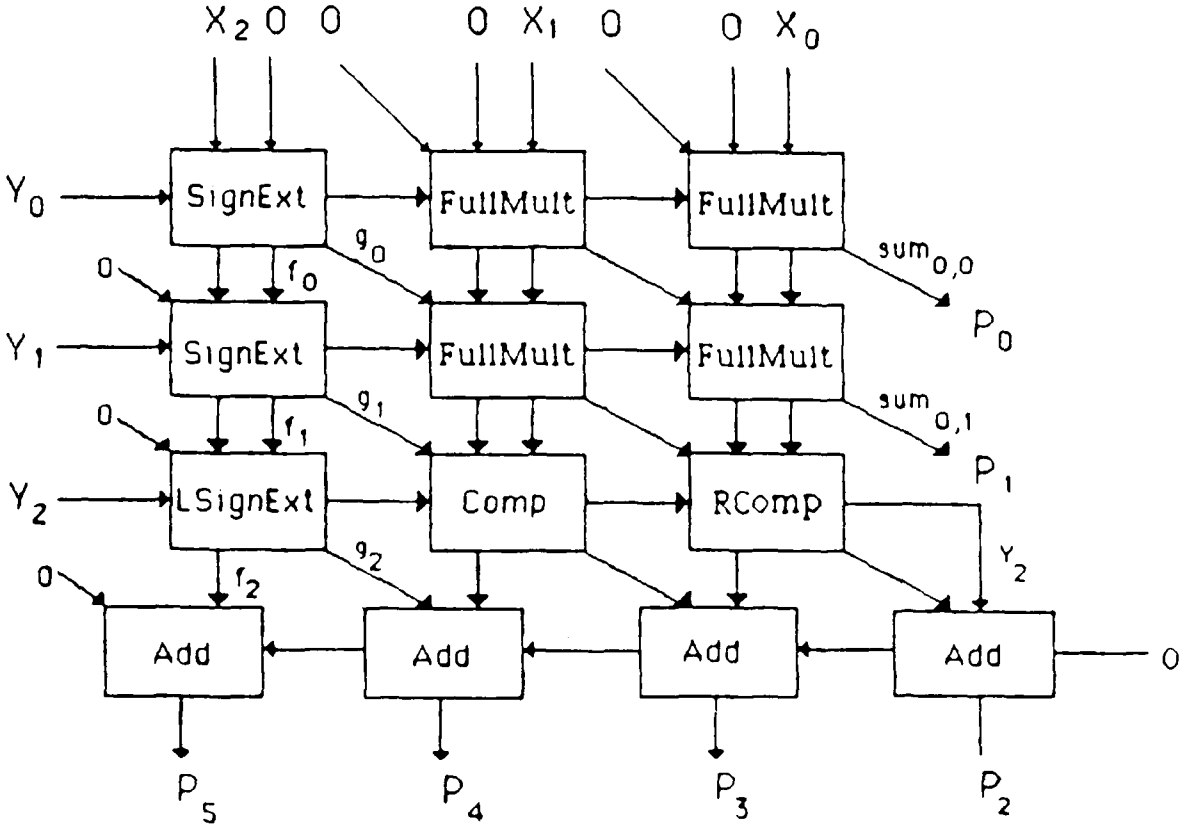


Figure 2-36: Schematic diagram of a 3 x 3 multiplier

2.3.2.2 The Functional Description

Consider now the functional description for an m x n signed two's complement multiplier. As mentioned before, the functional description describes the algorithm performed by a circuit. In the case of a multiplier, it describes how the final product is generated given a multiplicand X with *m* bits and a multiplier Y with *n* bits. Presented here are three versions of the functional description for a signed two's complement multiplier. Each version represents a different level of abstraction. At the highest level of abstraction, a 3 x 3 signed two's complement multiplier is described as follows.

```
NAME   multiplier;

TYPE   FUNCTIONAL;

PARAMETER   m = 3, n = 3;

MAIN

   multiplier = OUTPUT[l] (l = 0 .. m+n-1);

   INPUT = X[i] (i = 0 .. m-1),
           Y[j] (j = 0 .. n-1);

   OUTPUT = X * Y.
```

For an m x n multiplier, there are m+n bits in the final product. They are named OUTPUT[0], OUTPUT[1], ..., and OUTPUT[m+n-1]. The inputs are X and Y where X is of the form (X[m-1] X[m-2], ..., X[1] X[0]) and Y is of the form (Y[n-1] Y[n-2], ..., Y[1] Y[0]). The OUTPUT is the product of X and Y.

To further describe how the product output is generated, one lower level of description is provided.

```
NAME   multiplier;

TYPE   FUNCTIONAL;
```

```
PARAMETER   m = 3, n = 3;

MAIN

   multiplier = OUTPUT[l] (l = 0 .. m+n-1);

   INPUT = X[i] (i = 0 .. m-1),
           Y[j] (j = 0 .. n-1);

   OUTPUT = X * Y
          = CS[n];

   CS[k] = CS[k-1] + PP[k], if 1 ≤ k ≤ n;

   CS[0] = 0;

   PP[k] = Y[k-1] * X * 2**(k-1), if 1 ≤ k ≤ n;

   PP[0] = 0.
```

PP[k] represents the *k*th partial product and CS[k] represents the *k*th cumulative sum of the partial products. Note that the details of the extension of sign bit and the separation of carry and sum in each cumulative sum are not explicitly described in this level of abstraction. Examination of these statements shows that the functional description not only describes the algorithm but also corresponds to the schematic description, the mixed mode description and the layout description. The first n-1 rows of row[i] in the schematic description correspond to the first n-1 CS[k]'s while row[n] and the ripple adder are implicitly accounted for in the functional description by evaluating CS[n].

At the lowest level of abstraction, the generation of each bit of the product output and sign extension bit is described. It is shown as follows.

```
NAME   multiplier;

TYPE   FUNCTIONAL;

PARAMETER   m = 3, n = 3;
```

```
FUNC  sum, carry, summation;

MAIN

  /*  TERMINOLOGY:
    CS[k] = cumulative sum of partial products; made up of partial
            sum and partial carry. 1 ≤ k ≤ n.
    PS[k] = partial sum; the sum portion of the result of an addition
            when the carry overs are not rippled through the higher
            order bits.
    PC[k] = partial carry; the carry portion of the result of an
            addition when the carry overs are not rippled through
            the higher bits.
            Note: PS[k] + PC[k] = CS[k] the total result of the
            addition.
    F[k]  = the higher order bit resulting from extending the sign
            bit during an addition. This is also the MSB of PC[k].
    G[k]  = the lower order bit resulting from extending the sign
            bit during an addition. This is the MSB of PS[k].
    RS[l] = sum bit generated by the ripple adder. This is also
            one of the product bits. n-1 ≤ l ≤ m+n-1.
    RC[l] = carry bit generated by the ripple adder.
    PP[k] = partial product; it is one bit of Y times the vector X,
            then shifted appropriately.
    Fn[k,l] = bit with the 2**l power of the kth evaluation
            of the function Fn.     */


  multiplier = OUTPUT[l] (l = 0 .. m+n-1);

  INPUT = X[i] (i = 0 .. m-1),
          Y[j] (j = 0 .. n-1);

  OUTPUT = X * Y
         = RippleSum;

  RippleSum = summation (RS[t]*2**t, t = m+n-1 .. n-1) +
              summation (OUTPUT[t]*2**t, t = n-2 .. 0);

  RS[l] = sum (PS[n,l], PC[n,l], RC[l]), if n-1 ≤ l ≤ n+m-2;

  RS[m+n-1] = sum (0, PC[n,m+n-1], RC[m+n-1]);

  RC[l+1] = carry (PS[n,l], PC[n,l], RC[l]), if n-1 ≤ l ≤ n+m-2;
```

```
RC[n-1] = 0;

PC[n,n-1] = Y[n-1];

/* cumulative sum is made up of partial sum and partial carry */
/* PS[k] and PC[k] are not added until next addition */
CS[k] = PS[k] + PC[k], if 1 ≤ k ≤ n;

PS[k] = sum (PS[k-1], PC[k-1], PP[k])
      = summation (PS[k,t]*2**t, t = m+k-2 ..k) +
        summation (OUTPUT[t]*2**t, t = k-1 .. 0);

PS[0] = 0;

/* The MSB of a partial sum is G[k] */
PS[k,m+k-2] =  G[k];

PC[k] = carry (PS[k-1], PC[k-1], PP[k])
      = summation (PC[k,t]*2**t, t = m+k-1 .. k);

PC[0] = 0;

/* The MSB of partial carry is F[k] */
PC[k,m+k-1] = F[k];

F[k] = F[k-1] | PP[k-1,m+k-2], if 1 ≤ k ≤ n-1;

F[0] = 0;

G[k] = F[k-1] ^ PP[k-1,m+k-2], if 1 ≤ k ≤ n-1;

G[0] = 0;

F[n] = F[n-1] | (~X[m-1]*Y[n-1]);

G[n] = F[n-1] | (~X[m-1]*Y[n-1]);

PP[k] = Y[k-1]*X*2**(k-1)
      = summation (PP[k,t]*2**t, t = m+k-2 .. k 1), if 1 ≤ k ≤ n-1;

PP[n] = (X[m-1]*Y[n-1] - Y[n-1])*2**(m+n-2) +
      summation (Y[n-1]*~X[t]*2**(t+n-1), t = m-2 .. 0) + Y[n-1]*2**(n-1);

OUTPUT[l] = RS[l]: <timeing_spec>, if n-1 ≤ l ≤ n+m-1
          = PS[l+1,l]: <timing_spec>, if 0 ≤ l ≤ n-2.
```

Three functions, *sum*, *carry* and *summation*, are used in this description. *Sum* (a,b,c) is equal to **a XOR b XOR c** and *Carry* (a,b,c) is equal to **ab OR bc OR ac**. These two functions generate the sum out and carry out of three inputs. *Summation* is the addition of terms in the series. The range of the terms is specified by the limit of the index t. This function simulates the expansion of $\sum_{t=low}^{high} X_t$. Thus, *summation* (RS[t]*2**t, t = m+n-1 .. n-1) represents $RS[m+n-1]2^{m+n-1}+RS[m+n-2]2^{m+n-2}+\ldots+RS[n-1]2^{n-1}$. Figure 2-37 is the graphic representation of this functional description.

The lowest level of abstraction in the functional description is very close to the algorithm. RippleSum is the final product which is the concatenation of the sum bits (RS[m+n-1]RS[m+n-2]...RS[n]RS[n-1]) generated by the ripple adder and the *n*-1 bits generated by the partial sum's, PS[n-1],PS[n-2],...,PS[1]. In the ripple adder, each of the sum bits (which is also one bit of the final product) is defined as the *sum* part of three components: (1) one bit of the partial sum from the *n*th cumulative sum, (2) one bit of the partial carry from the *n*th cumulative sum, and (3) the carry from its next lower order bit. Similarly, every bit of a partial sum is the *sum* part of the sum in, the carry in and one bit of the current partial product. The carry part of these components forms one bit of the partial carry. These can be seen very easily from Figure 2-37. It should be noticed that the the MSB of each partial sum (PS[k]) is G[k] and the MSB of each partial carry (PC[k]) is F[k]. The definitions of F[k] and G[k] are the same as those described in Section 2.3.1.2. By separating the cumulative sum (CS[k]) into partial sum and partial carry, the carry is not rippled to the next higher order bit. Thus, the multiplication is a sequence of carry-save additions and only suffers from one ripple addition at the end. The third level of abstraction in the functional description reflects this important idea as it is implemented in the generator.

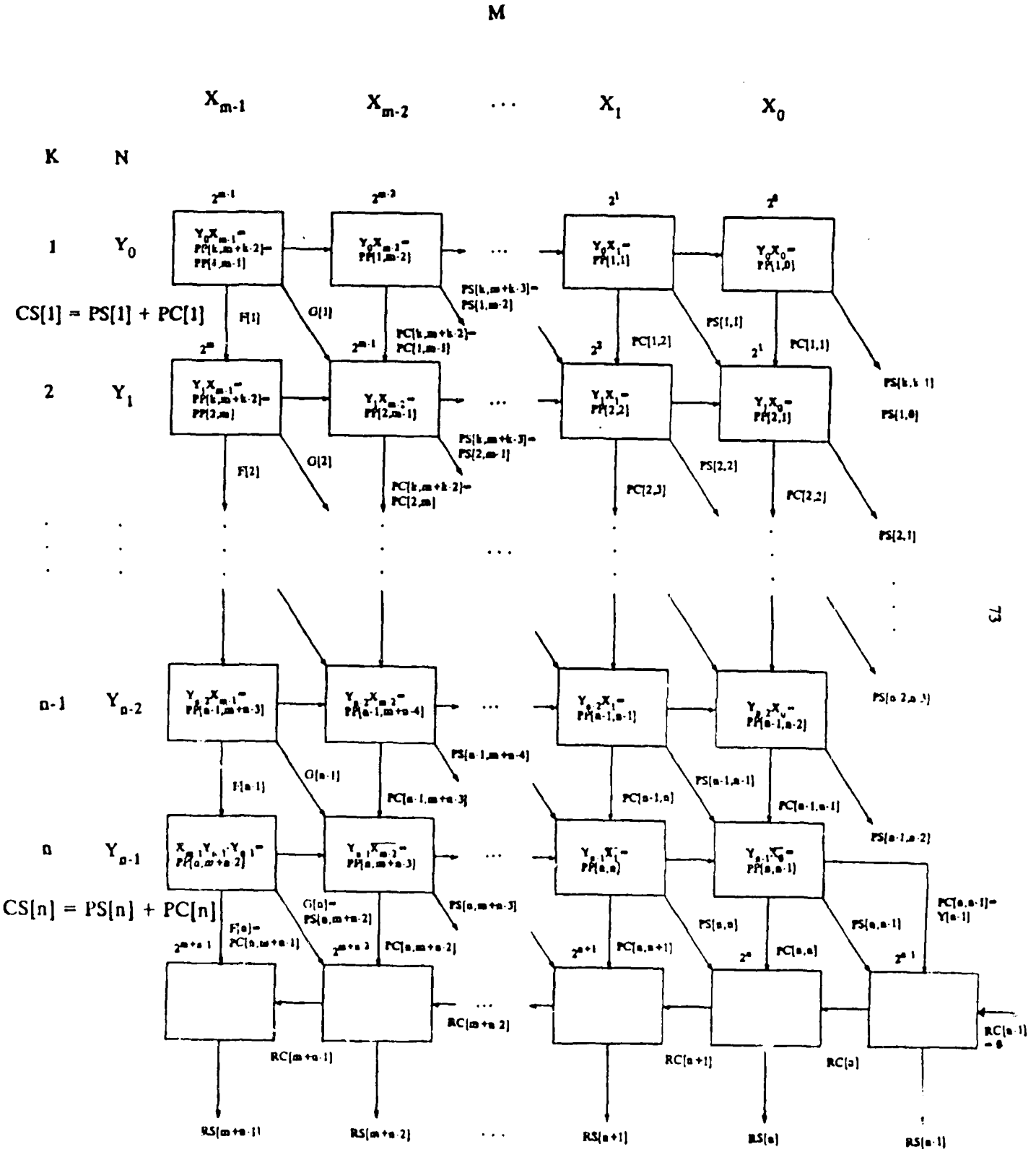


Figure 2 .7: Graphic representation of the functional description

As in the decoder example, we see that each description employs many levels of abstraction and that there is a correspondence among different descriptions. If the design is changed, relevant parts for a description can be discovered and modified very easily. The changes can also propagate to other descriptions in a controlled manner. The process of design and documentation is thus enhanced.

CHAPTER 3. PARAMETERIZATION

Chapter 2 shows how a set of notations can be used to describe the multiple equivalent representations of a design. Each description is *declarative*. A circuit is described in the form of a hierarchy. As a result, the abstract structure of a design is captured. Each description describes an *instance* of a particular circuit. The flexibility of specifying instances with different design specifications is needed. This chapter discusses the parameterization issues. Section 3.1 presents how the description is modified to describe instances with different parameters. Section 3.2 suggests a structure which contains information about parameters (attributes) of an instance of a circuit. This structure, **catalog**, serves as a database such that important properties of a circuit can be retrieved by the user or the interface system without expanding the relevant description. Finally, section 3.3 shows the robustness of the descriptions when one of the parameters, technology, is changed.

3.1 Instances with Different Attributes

Using the constructs described in the last chapter, a decoder with a different number of select wires and a multiplier with a different number of bits in the multiplier and the multiplicand can be specified by providing the desired value(s) of n (and m, in the case of multiplier) in the PARAMETER declaration part. For instance, the four different views of descriptions for a NAND style decoder with 5 select wires are the same as those given in section 2.2 except that the value of n is assigned to be 5 (n = 5) rather than 3. Thus, without specifying the values of n, the descriptions given in section 2.2 can serve as generic descriptions for *any* NAND style decoder. When a description is instantiated, the value of n is bound to a specific number. In other words, each occurrence of n in the description is replaced by the desired number of select wires. The description for a particular

instance of decoder is thus accomplished. This flexibility through the use of input parameters simplifies the descriptions for a class of circuits which have different dimensions in their inputs but have similar structures. Parameterization of input helps control the variability in their dimensions. Moreover, one can see how the description varies with the parameters.

The concept of parameterization can be extended to the LEAF CELLS declaration in the layout description, the mixed mode description, and the functional description. Given a set of design specifications, a circuit description is instantiated. The appropriate leaf cells for that description must be given in the LEAF CELLS declaration. Notice that the leaf cells may not be the same for different specifications; however, the hierarchical structures of objects in the description are similar. Suppose that instead of a 3 x 3 *signed* two's complement multiplier we want to generate the schematic description for a 3 x 3 *unsigned* multiplier. This can be accomplished by the following statements:

```
NAME   multiplier;
TYPE   SCHEMATIC;
PARAMETER   m = 3, n = 3;
LEAF CELLS   FullMult, Add;
MAIN
    multiplier = adder | (| row[i] (i = n .. 1));
    row[i] = -- FullMult (m);
    adder = -- (Add (m)).
```

The schematic diagram is shown in Figure 3-1.

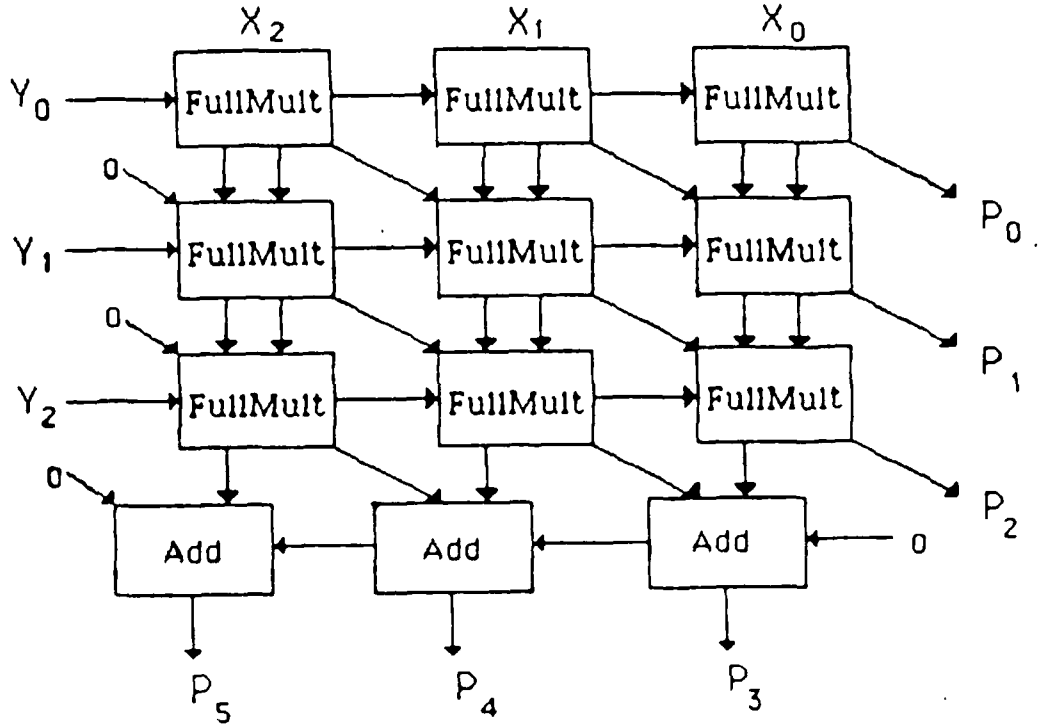


Figure 3-1: Schematic diagram of a 3 x 3 unsigned multiplier

The description of the unsigned multiplier is simpler than the one for the two's complement multiplier. There is a similarity between the hierarchical structures of objects in these two descriptions. Figures 3-2 and 3-3 show the hierarchical structure of objects in the signed two's complement multiplier and the unsigned multiplier, respectively.

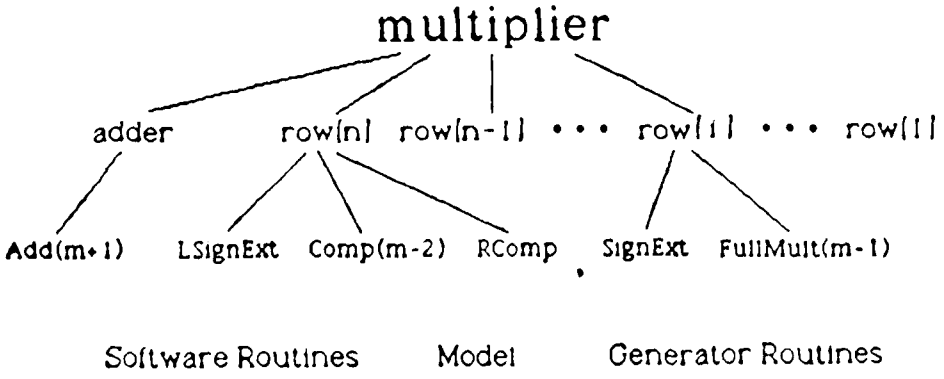


Figure 3-2: Hierarchical structure of signed 2's complement multiplier

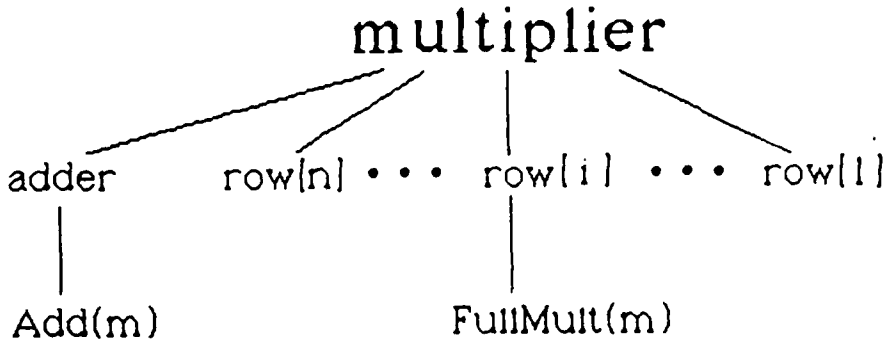


Figure 3-3: Hierarchical structure of unsigned multiplier

At the highest level of both descriptions, adder and rows serve as primitive objects which encapsulate internal details of the circuit so that the multiplier can be described in a more comprehensible way. Then the rows and adder are further defined at one level lower in the hierarchy. Because the most significant bits of the multiplicand and the multiplier in the unsigned multiplier also carry positive magnitude, the definition of row[n] is the same as the one for other rows and each row only consists of m instances of FullMult.

The functional description for a 3 x 3 unsigned multiplier corresponds to the schematic description and bears a resemblance to the functional description for a 3 x 3 signed two's complement multiplier. The first two levels of abstraction are the same as those in the signed two's complement multiplier. Since both the multiplicand and the multiplier are non-negative, there is no need for complementing X in the last partial product and performing sign bit extension for each cumulative sum. As a result, the third level of abstraction is simpler. The following statements illustrate the functional description for a 3 x 3 unsigned multiplier at the third level of abstraction.

```
NAME  multiplier;

TYPE  FUNCTIONAL;
```

```
PARAMETER   m = 3, n = 3;

FUNC    sum, carry, summation;

MAIN

   multiplier = OUTPUT[l] (l = 0 .. m+n-1);

   INPUT = X[i] (i = 0 .. m-1),
           Y[j] (j = 0 .. n-1);

   OUTPUT = X * Y
          = RippleSum;

   RippleSum = summation (RS[t]*2**t, t = m+n-1 .. n) +
               summation (OUTPUT[t]*2**t, t = n-1 .. 0);

   RS[l] = sum (PS[n,l], PC[n,l], RC[l]), if n ≤ l ≤ n+m-2;

   RS[m+n-1] = sum (0, PC[n,m+n-1], RC[m+n-1]), if n ≤ l ≤ n+m-2;

   RC[l+1] = carry (PS[n,l], PC[n,l], RC[l]), if n ≤ l ≤ n+m-2;

   RC[n] =  0;

   CS[k] = PS[k] + PC[k], if 1 ≤ k ≤ n;

   PS[k] = sum (PS[k-1], PC[k-1], PP[k])
         = summation (PS[k,t]*2**t, t = m+k-2 .. k) +
           summation (OUTPUT[t]*2**t, t = k-1 .. 0);

   PS[0] = 0;

   PC[k] = carry (PS[k-1], PC[k-1], PP[k])
         = summation (PC[k,t]*2**t, t = m+k-1 .. k);

   PC[0] = 0;

   PP[k] = Y[k-1]*X*2**(k-1)
         = summation (PP[k,t]*2**t, t = m+k-2 .. k-1), if 1 ≤ k ≤ n;

   OUTPUT[l] = RS[l]: <timing_spec>, if n ≤ l ≤ n+m-1
             = PS[l+1,l]: <timing_spec>, if 0 ≤ l ≤ n-1.
```

In this section, we have examined some instances of decoder and multiplier with attributes different from those specified in chapter 2 . The intent was to show how parameters are used to modify the descriptions, and yet the object hierarchies remain consistent across the same family of circuits. Use of hierarchy and abstraction in the circuit description makes it possible to suppress unnecessary details. However, important information about a circuit after it is "expanded" should also be provided to the user and the applications software to facilitate the design process. **Catalog**, a database for an instantiated circuit, is designed to serve this function and will be discussed in the next section.

3.2 Catalog

A catalog is a list of properties of an *instantiated* circuit. It provides information about the expanded version of a circuit to the user and applications software, such as the layout system and to the interconnect system. Especially, features which are not shown in the declarative descriptions but are requisites for the generation of layout or simulation are provided. In the declarative descriptions, the multiple instantiation of an object is specified by repetition. Since the details of an object are only considered once, this produces considerable savings in design. Nevertheless, the characteristics of the expanded circuit such as size and bus width are often requested by the synthesis tools (e.g. placement), simulators, DRC interface, or timing analyzer. The objective of a catalog is to permit the users and the interface systems to *retrieve* attributes of the circuit from the database rather than *analyze* the descriptions or the expanded geometric representations. Thus fast access of information is achieved.

The entries of a catalog are determined by the circuit and the interface systems. They can be extended. A catalog has either *general* or *special* entries. General entries exist in all catalogs while special entries are included only in some

circuits. The entries of a catalog may include the following. Item (1) to item (7) are general entries while item (8) represents special entries.

(1) *Name*: the name of the circuit, e.g. decoder. So, the value of the type is string.

(2) *Description*: a description of the circuit behavior. This description is primarily for documentation purposes. It can be an English-language description or a pointer which points to a file that contains this information.

(3) *Technology*: e.g. cmos 3 micron, nmos, etc. Again, the value of the type is a string.

(4) *Parameters*: number of inputs and outputs. These are integers. In addition, other design specifications should also be included. Examples are the style (NAND or NOR) of a decoder and the number representation (SIGNED or UNSIGNED) of a multiplier. These parameters are circuit dependent and their values depend on the nature of the parameters.

(5) *Size*: the X and Y dimensions of the bounding box which contains the circuit. They can be expressed in terms of lambdas or microns.

(6) *Connectivity*: The border descriptions include lists of coordinates of the points (say, in clockwise order) where each kind of material in the circuit makes contact with the bounding box. They are provided for routing purposes. Information about the location and width of the power/ground buses and the wires in the design should also be included. For each wire, it contains the wire id, its layer, its size and location. The values of the border descriptions and wire information in the catalog can be specified by pointers which point to the appropriate files. Moreover, the catalog should also provide a pointer which points to the DRC ring specifications.

(7) *Performance*: gate size, power consumption, rise time, fall time and propagation of critical path.

(8) *Special Features*: characteristics that are only applicable to some circuits. For instance, a bit map specification is useful in a ROM while it is irrelevant to a decoder.

The structure of a catalog can be viewed as a collection of tables which are similar to the relations in the relational database. Because some of the entries in a catalog are circuit dependent, no attempt is made here to present all possible relations. Part of the structure is shown as follows.

(1) *circuit (name, description, technology, parameters, size, connectivity, performance, special features)*. Each circuit is associated with a description of its name, behavior description, technology, parameters, size, connectivity, performance, and special features. The entries for the last five attributes are pointers. Since we are describing *an* instantiated circuit, there is only one record in the circuit table. However, if "catalog" is generalized to represent a generic construct for all the circuits in a common family, then there will be sets of records in the circuit table and each record represents a circuit.

(2) *parameters (#inputs, #outputs, ...)*.

(3) *size (Xdimension, Ydimension)*.

(4) *connectivity (crossings, wires, drcRing)*. The values of attributes in the connectivity table are all pointers.

(5) *crossings (side, layer, location)*. The crossings table describes the crossings in the border description. It contains one line per crossing, representing the containing side (top, down, left, right), its layer and location.

(6) *wires (id, layer, size, location).*

.
.
.

Information about the components of a circuit and how they are oriented and placed is not described in the catalog since they are already specified in the declarative descriptions. The catalog documents the expanded version of the circuit and makes connectivity information explicit. So, the users and the applications software can query it. In conjunction with the declarative descriptions and leaf cells, they are used to guide the generation process of a circuit.

3.3 Change of Technology

One of the desired properties of the declarative descriptions is technological independence. The descriptions of a circuit should be invariant to the change of the technology. The only changes that a designer has to make are the innards of leaf cells, since they are primitive components in the description hierarchy and they embed the technology dependent characteristics (i.e. implementation details such as the layers used to fabricate the layout) of a circuit.

A small experiment was conducted to check whether the layout description for a NAND style decoder holds when the technology is changed from 3 micron fabricator to 1.2 micron fabricator. The dimensions of each box and distance between any two materials within each of the leaf cells are carefully checked and modified manually according to the design rules for 1.2 micron fabricator. It is found that there is no significant difference between the new leaf cells and those for the 3 micron fabricator. Since the implementation of the leaf cell is local to the cell and variation in the implementation does not functionally affect the other components in the hierarchical descriptions, it is obvious that the declarative description holds regardless of the technology used.

CHAPTER 4. CONCLUSIONS

This chapter summarizes the contributions of this thesis towards the model construction in the VLSI Design Generators project. Section 4.1 summarizes the main features of the declarative descriptions and the parameterization issues. The contributions of this thesis must be viewed in the context of design and documentation. Section 4.2 gives some directions for future research.

4.1 Summary

In this thesis, the need of a model in the circuit generation process has been described. VLSI design is an inherently complex activity, made even more complex by the need to maintain several representations of an object being designed. This is because different design tools have different requirements. The multiple representation problems were discussed. It is observed that the model should provide descriptions of the multiple equivalent representations of a circuit at the optimal level of abstraction for design and documentation purposes.

An integrated circuit design can be described in several forms, among which are its layout, its transistor schematic, its logic gate schematic, and its functional behavior. A set of notations to be used in the various descriptions were introduced. Their syntax as well as semantics were discussed. These notations were created to make the descriptions simple, natural, expressive, and to show abstract, hierarchical structure and technology independence. Two examples, a decoder and a multiplier, were used to illustrate the application of the descriptions.

For each design, there are some common characteristics among the four different views of description. They are

- The descriptions are declarative.

- The hierarchical decomposition of the design proceeds recursively. Multiple levels of abstraction make it possible to suppress unnecessary details and make the design more comprehensible.
- Substitution is the mechanism to navigate in the hierarchical description.
- There is a correspondence between different descriptions. As a result, the design changes can be propagated in a controlled manner across different descriptions.

Because of these characteristics, the declarative descriptions contributes to the design process. A circuit can be developed faster and debugged easier since it is specified in terms of a simple high-level description. The hierarchical structure of the description reduces the complexity of a design, which is a big asset in the VLSI design environment. The descriptions also serve as a choice vehicle for the documentation of a design. They are simple and expressive. Most of all, they are abstract enough to suppress the unnecessary details of a circuit while still make the complex data structure explicit. This helps the designer review his design decisions and communicate his ideas to others easily.

Each of the descriptions has its specific function in the generation process of a design. The layout description in conjunction with an appropriate set of leaf cells and the catalog can be used to generate the layout of a circuit. The mixed mode description helps create the logic network description for the NETLIST program and simulation. The schematic description simplifies the construction of the mixed mode description. The functional description serves as an excellent reference for checking the intermediate results for the circuit simulation.

Finally, the parameterization issues were discussed to show the flexibility of the description. For different instances of a circuit, parameters are used to modify the descriptions, and yet the object hierarchies remain consistent. In addition, the leaf cells may vary with the technology used while the descriptions for a design are invariant. The purpose of a catalog was reviewed, and the candidate entry types

were presented. It is believed that the incorporation of the declarative descriptions, the catalog, and a collection of leaf cells can capture the complex data structure of a design and greatly facilitate the VLSI generation process.

4.2 Future Work

The work presented in this thesis forms a solid foundation for the development of the generator construction methodology. Various extensions to the declarative descriptions and the catalog can be done to form a more powerful and flexible model that guides the generation process. The following are brief descriptions of three areas of research which need further exploration.

1. *building a translation system which can generate the appropriate outputs for different descriptions of a circuit.*

From the layout description, the mixed mode description, and the schematic description, we should be able to derive the layout (together with the caesar or CIF file), the transistor diagram and the logic gate diagram of a design, respectively. The geometric operators in the declarative descriptions only describe the relative placement of leaf cells, and the actual locations of alignment are embedded in the registration marks. The most promising and challenging work to be done is to build a system which can align cells according to the registration marks on the relevant edges. The geometric operators indicate the edges on which relevant registration marks can be found. The two objects will be aligned so that their corresponding registration marks are adjacent to each other, horizontally or vertically, according to the relations specified by the geometric operators. Also research needs to be done on connecting two cells which are not adjacent to each other.

Presently, the generation of the leaf cells in the layout description is done by

CAESAR and CFL. In the case of the mixed mode description and the schematic description, *postscript* code is a good candidate for generating the leaf cells.

2. *investigating ways of describing the catalog database.*

Parenthesized expressions of the form *(Attribute value 1 value 2 ...)* can be used for writing down the catalog database description. The LISP-like format is easy to extend with new attributes. Another approach is to prompt the user tables which list all the general entries on one column. The user is expected to type in the appropriate values of these entries on another column according to the particular characteristics of the circuit. The special entries may also be given by tables or provided by a different mechanism. This approach is more user friendly, but requires a more sophisticated system.

3. *creating a design database which can organize the design data across the multiple representations of a design.*

A circuit design can be described simultaneously by its layout, interconnected transistors, interconnected logic gates, and functional behavior. Each description is a hierarchical collection of objects. To keep the design description consistent within and across representations, it is useful to have a database which can organize the object hierarchy within each description, correlate equivalent objects across the multiple descriptions, and maintain these correspondences as the design changes. As a result, if a portion of the design is changed in one representation, the system can flag the corresponding portions in other representations of the circuit, and appropriate modifications can be made easily.

BIBLIOGRAPHY

[Bamji 85] Bamji, C., Hauck, C. and Allen, J.
A Design by Example Regular Structure Generator.
In *22nd Design Automation Conference*, pages 16-22. IEEE, 1985.

[Chu 84] Chu, K. and Lien, Y.
Database Concepts in the VDD System.
A Quarterly Bulletin of the IEEE Computer Society Technical Committee on Database Engineering 7(2), June, 1984.

[Clarke 85] Clarke, E. and Feng, Y.
Escher--A Geometrical Layout System for Recursively Defined Circuits.
Research Report CMU-CS-85-150, Department of Computer Science, Carnegie-Mellon University, July, 1985.

[Ellis 81] S. Ellis.
A Symbolic Layout Language & Database for an Integrated VLSI Design System.
Master's thesis, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, December, 1981.

[German 85] German, S. and Lieberherr, K.
Zeus: A Language for Expressing Algorithms in Hardware.
IEEE Computer :55-65, February, 1985.

[Katz 82] Katz, R.
DAVID: Design Aids for VLSI Using Intergrated Databases.
A Quarterly Bulletin of the IEEE Computer Society Technical Committee on Database Engineering 5(2), June, 1982.

[Katz 83a] Katz, R.
Chip Assemblers: Concepts and Capabilities.
In *20th Design Automation Conference*, pages 25-30. IEEE, 1983.

[Katz 83b] Katz, R.
Managing the Chip Design Database.
IEEE Computer :26-36, December, 1983.

[Katz 85a] Katz, R.
Computer-Aided Design Databases.
IEEE Design & Test :70-74, February, 1985.

[Katz 85b] Katz, R.
Information Management for Engineering design.
Springer-Verlag Berlin Heidelberg, 1985.

[Lipton 82] Lipton, R., North, S., Sedgewick, R., Valdes, J., and Vijayan, G.
ALI: A Procedural Language to Describe VLSI Layouts.
In *19th Design Automation Conference*, pages 467-474. IEEE, 1982.

[Sheeran 83] Mary Sheeran.
μ FP - An Algebraic VLSI Design Language.
PhD thesis, Oxford University Computing Laboratory, November, 1983.

[Suzuki 85] Suzuki, N.
Concurrent Prolog as an Efficient VLSI Design Language.
IEEE Computer :33-40, February, 1985.

[UW/NW 84] UW/NW VLSI Consortium.
Quality VLSI Design Generators.
1984
A Research Proposal Submitted to The Defense Advanced Research Projects Agency Information Processing Technology Office by Department of Computer Science's University of Washington/Northwest VLSI Consortium.

[UW/NW 85] UW/NW VLSI Consortium.
VLSI Design Tools -- Reference Manual
1985.
Release 3.0 , TR# 85-07-03.

[Winder 84] Wayne Winder.
Design Note of the MULT generator.
1984

APPENDIX A. EBNF DEFINITION

In the definition given below, the equal sign "=" is to be read as "is defined as". All literals are enclosed in double quotes (i.e., " ") while <char>, <letter>, and <digit> represent the sets of characters, letters, and digits, respectively. **Concatenation** is expressed by writing terms (factors) followed by another. The vertical bar "|" is used to separate **alternatives** in the definitions. Braces, i.e., "{" and "}", specify 0 or more **repetitions**, and brackets, "[" and "]", express **options**. Each definition ends with a ".".

<program> = <declaration> "MAIN" <statement> [";" <in-spec>]
{";" <statement>}.

<declaration> = <name-decl> <type-decl> <param-decl> [<cell-decl>]
[<func-decl>].

<name-decl> = "NAME" <name> ";".

<name> = <letter> {<letter> | <digit> | <"-">}.

<type-decl> = "TYPE" <type-group> ";".

<type-group> = "LAYOUT" | "MIXED" | "SCHEMATIC" | "FUNCTIONAL".

<param-decl> = "PARAMETER" <param> {"," <param>} ";".

<param> = <name> "=" <integer>.

<integer> = <digit> {<digit>}.

<cell-decl> = "LEAF CELLS" <name> {"," <name>} ";".

<func-decl> = "FUNC" <name> {"," <name>} ";".

<statement> = <regular-statement> | <out-spec>.

<regular-statement> = <object> "=" <body> {"=" <body>}.

<object> = <name> ["[" <index-list> "]"].

<index-list> = <expr> {"," <expr>}.

<expr> = ["+" | "-"] <term> {<op1> <term>}.

<term> = <factor> {<op2> <factor>}.

<factor> = integer | ["~"] <object> | "(" <expr> ")" |
["~"] <simple-func> | <complex-func>.

<op1> = "+" | "-" | "|".

<op2> = "*" | "/" | "**" | "%" | "&" | "^".

<simple-func> = <name> "(" <object> {"," <object>} ")".

<complex-func> = <name> "(" <expr> "," <subrange> ")".

<subrange> = <name> "=" <expr> ".." <expr>.

<body> = <assignment> ["," <IfCond>].

<assignment> = <expr> | <out-assign> | <geo-assign>.

<out-assign> = "**OUTPUT**" <in-out>.

<in-out> = "[" <name> "]" "(" <subrange> ")".

<in-spec> = "**INPUT**" "=" <input> {"," <input>}.

<input> = <name> <in-out>.

<out-spec> = "**OUTPUT**" ["[" <index-list> "]"] "=" <out-body> {"=" <out-body>}.

<out-body> = <assignment> ":" <expr> ["," <IfCond>].

<geo-assign> = <comp> {<geo-op1> <comp>}.

<geo-op1> = "--" | "|" | "-- ∩" | "| ∩".

<comp> = <object> | <geo-op2> "(" <object> ["," ["+"|"-"] <integer>] ")" |
"(" <loop> ")".

<geo-op2> = "**rot**" | "**mx**" | "**my**".

<loop> = <geo-op1> "(" <object> "(" <loop-index> ")" ")".

<loop-index> = <expr> | <sub-range> ["," <expr>].

<IfCond> = "IF" <relation> { ["&&" | "||"] <relation>}.

<relation> = <expr> <re-op> <expr> [<re-op> <expr>].

<re-op> = "<=" | "<" | "==" | "!=" | ">" | ">=".

APPENDIX B. LEAF CELLS IN THE DECODER LAYOUT

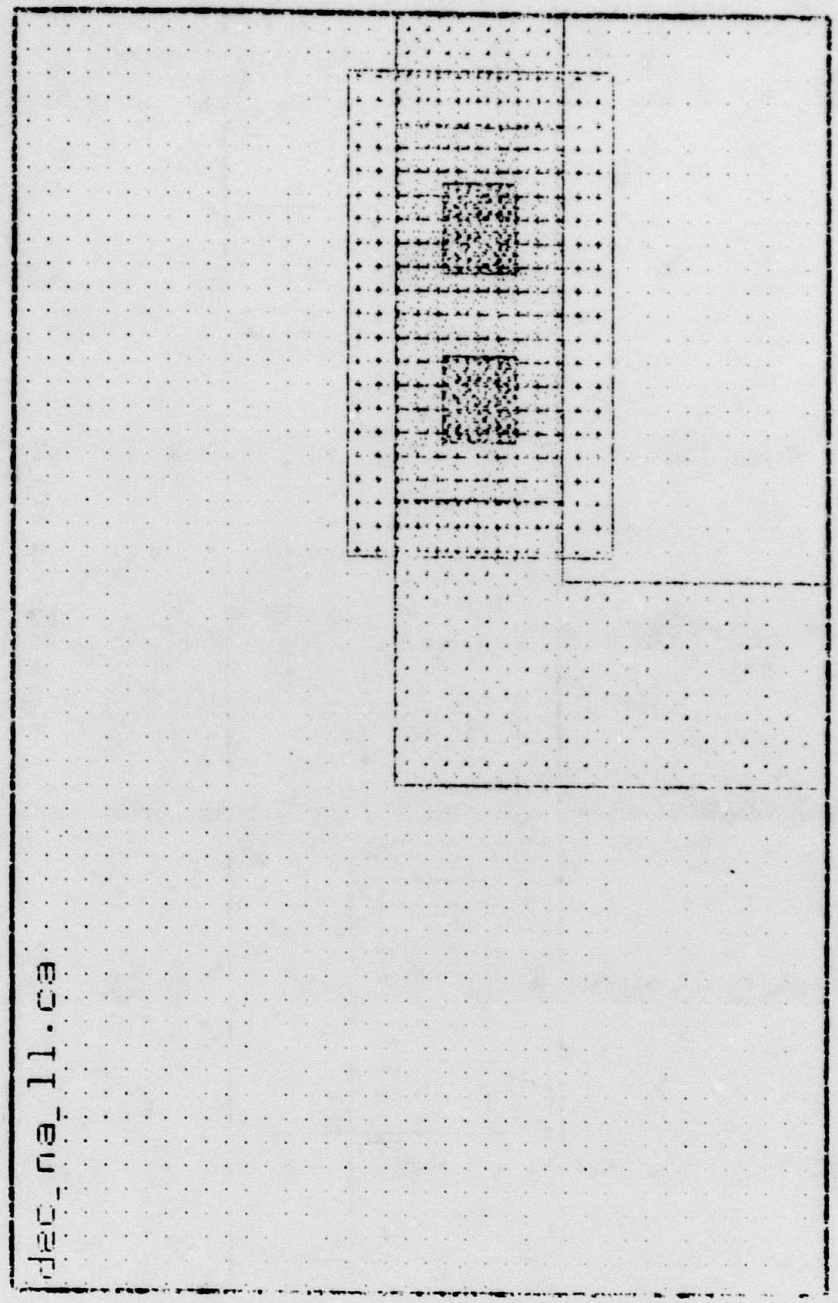
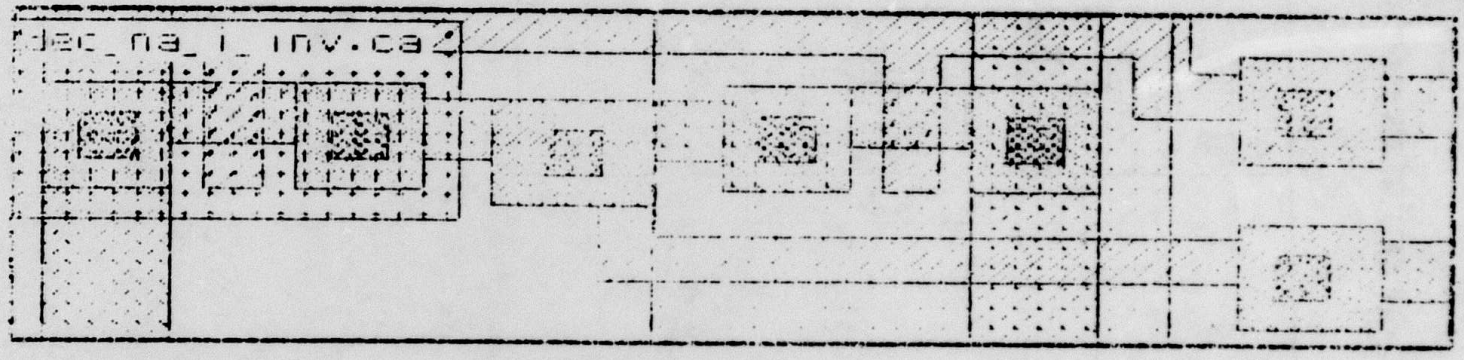


Figure B-1: dec_na_ll



Scale: 1 micron is 0.05 [illegible]

Figure B-2: dec_na_i_inv

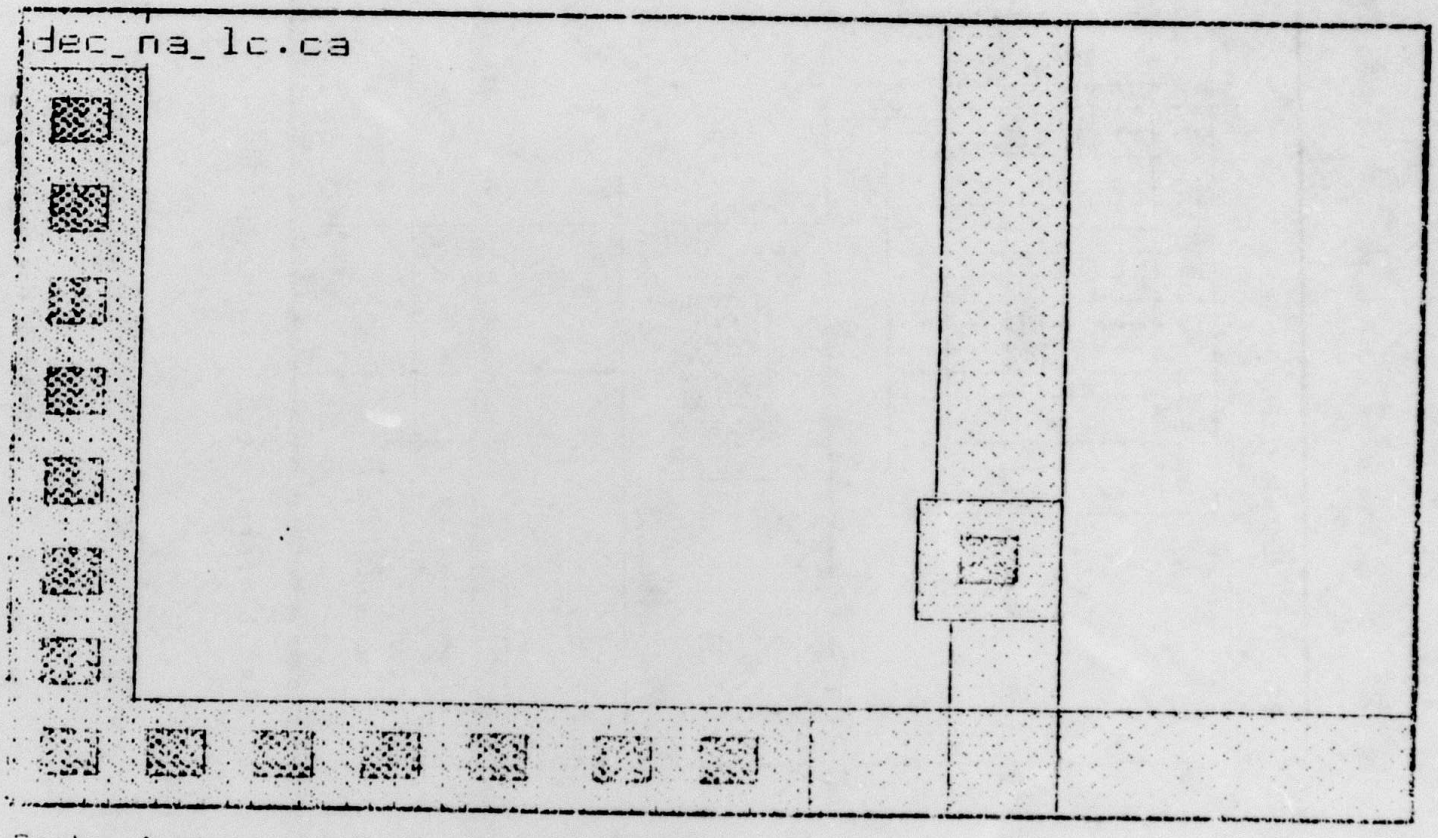


Figure B-3: dec_na_lc

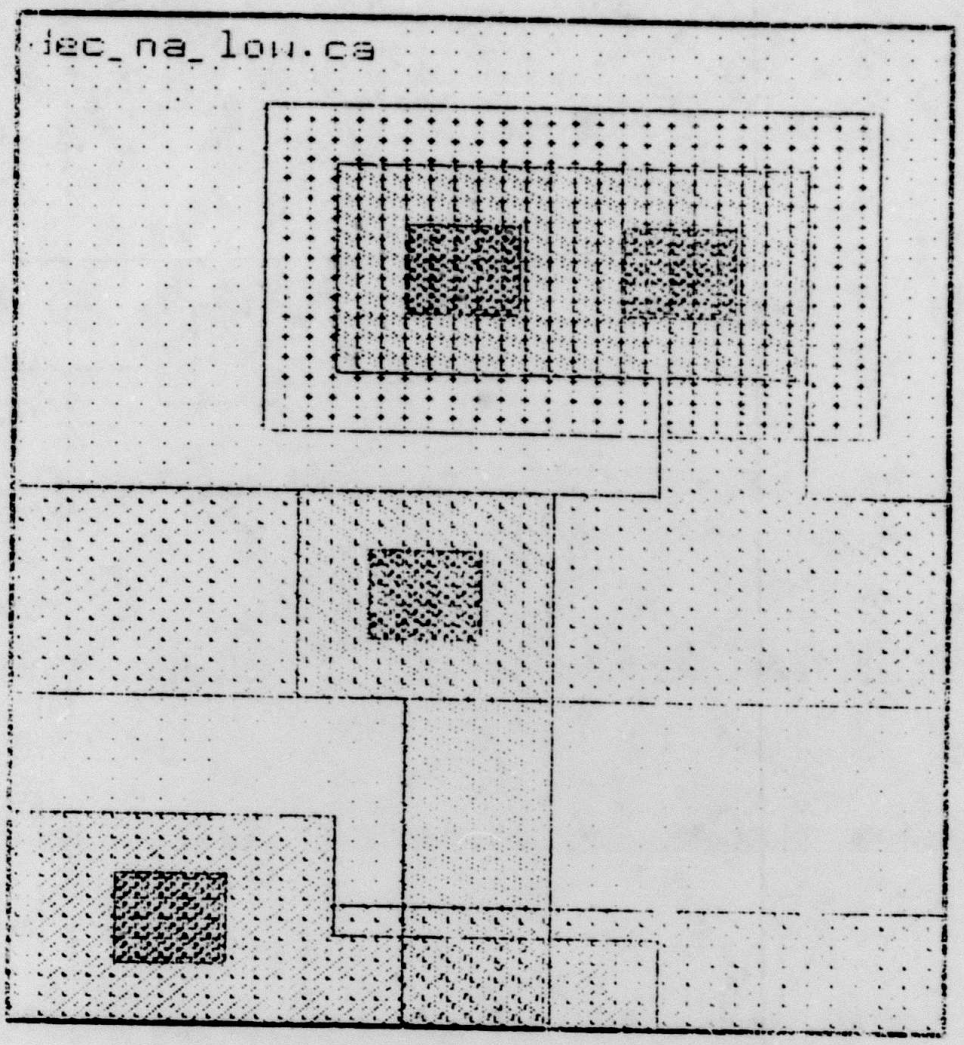


Figure B-4: dec_na_low

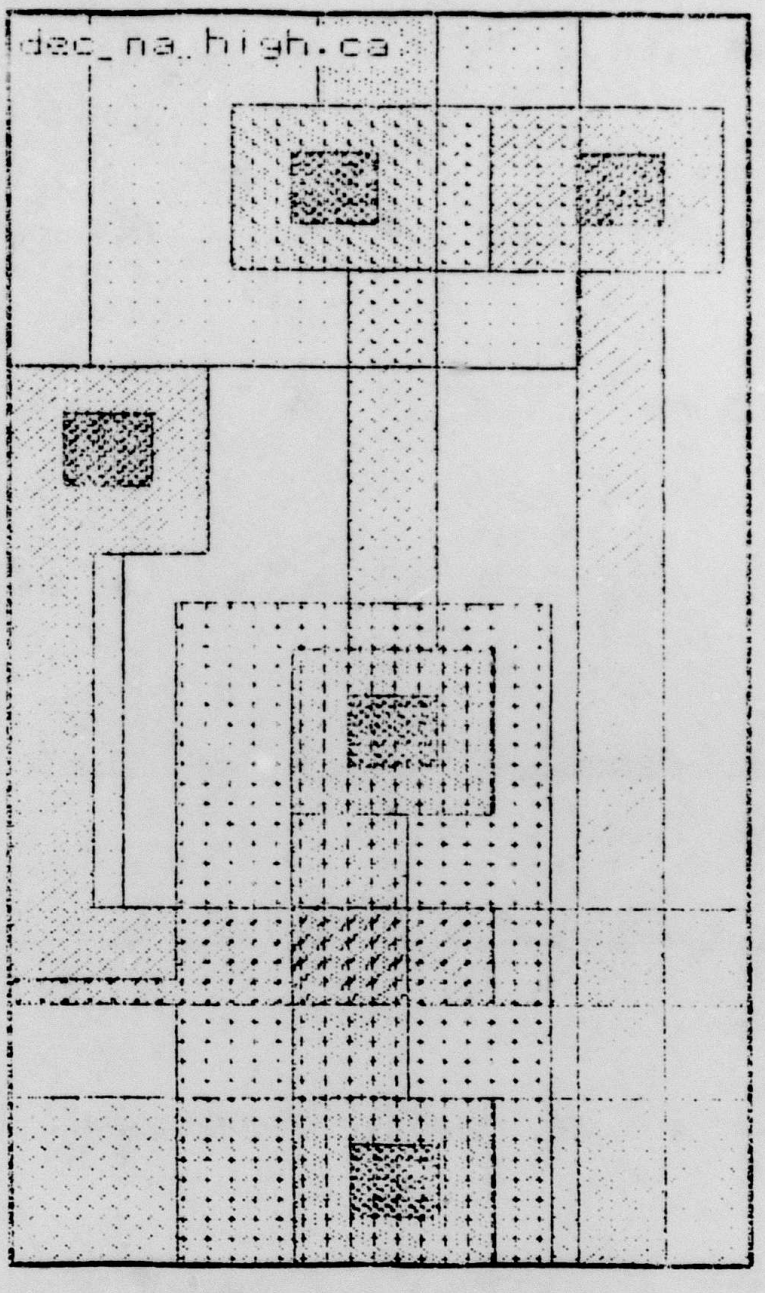


Figure B-5: dec_na_high

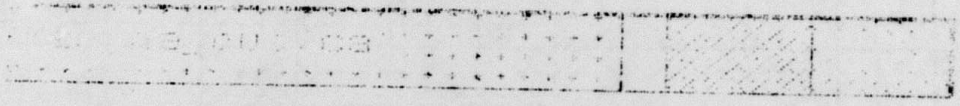


Figure B-6: dec_na_out

APPENDIX C. LAYOUT DESCRIPTION OF A MULTIPLIER

This is the layout description of a 3 x 3 signed multiplier. In this description, *G* and *V* are defined as follows.

G = ((*horiz_bus_gnd* - 1) DIV MIN_BUS) + 1
V = ((*vert_bus_vdd* - 1) DIV MIN_BUS) + 1

where MIN_BUS is designer-defined; *horiz_bus_gnd* and *vert_bus_vdd* are determined by *m* (number of bits in the multiplicand) and *n* (number of bits in the multiplier). The left side bus is GND.

```
NAME   multiplier;

TYPE   LAYOUT;

PARAMETER   m = 3, n = 3;

LEAF CELLS
            MULTMU, MULTMG, MULTMD, MULTEU, MULTEG,
            MULTED, MULTLU, MULTLG, MULTLD, MULTCU,
            MULTCG, MULTCD, MULTXA, MULTXG, MULTXO,
            MULTXV, MULTCPM, MULTYA, MULTYG,
            MULTYO, MULTASILFN, MULTAMIDN, MULTASIRTN,
            GNDarea, Edge, GNDend, VDDend, Right_Corner,
            Out_Path, Polystrip;

MAIN

      multiplier = ( ( (GNDarea -- Mult_X)
                  |∩ (Mult_Y -- Array) ) -- VDDarea )
                  |∩ Adder;

      Array = (|∩ (row[i] (n-1))) |∩ row[n];

      row[i] = sign --∩ (--∩ (fullmult (m-1) ) );

      row[n] = lastsign --∩ (--∩ (complement (m-1) ) );
```

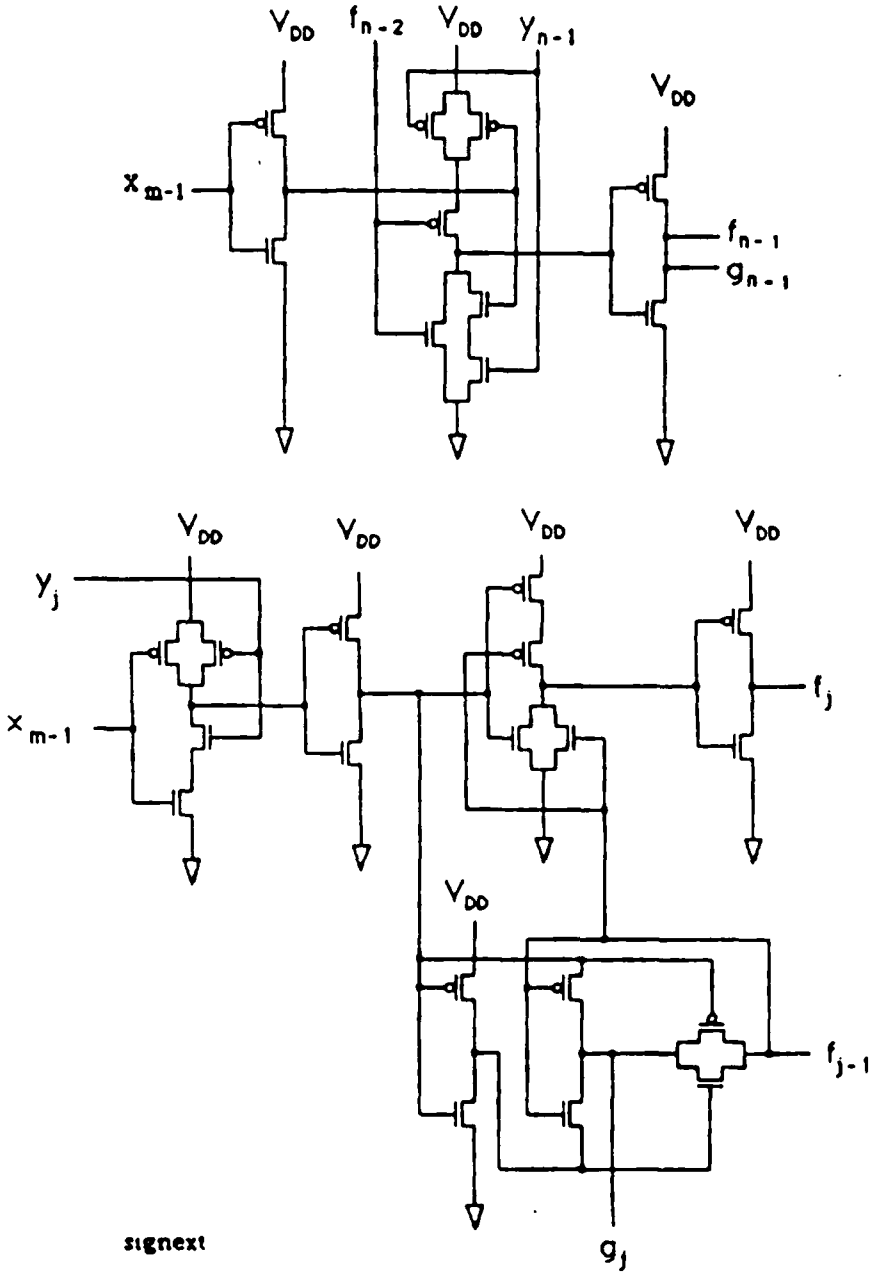
```
fullmult = MULTMU ∩ (∩ (MULTMG (G)) ) ∩ MULTMD;

sign = MULTEU ∩ (∩ (MULTEG (G)) ) ∩ MULTED;

lastsign = MULTLU ∩ (∩ (MULTLG (G)) ) ∩ MULTLD;

complement = MULTCU ∩ (∩ (MULTCG (G)) ) ∩ MULTCD;

Mult_X = (--∩ (X_Comp (m)));

X_Comp = rot(MULTCPM, -90) ∩ MULTXA ∩ MULTXG ∩
         MULTXO ∩ (∩ (MULTXV (V)));

Mult_Y = (∩ (Y_Comp (n)));

Y_Comp = MULTCPM -- Edge -- MULTYA --∩ MULTYG
         --∩ MULTYO;

Adder = GNDend -- add -- VDDend;

add = MULTASILFN -- (-- (MULTAMIDN (m-1)) ) -- MULTASIRTN;

VDDarea = Right_Corner ∩ (∩ (Out_Path (n-1))) ∩ (Polystrip).
```

APPENDIX D. MIXED MODE DESCRIPTION

The mixed mode description for a 3 x 3 signed multiplier is as follows. The expansion of each of the leaf cells is shown after that.

```
MANE   multiplier;
TYPE   MIXED;
PARAMETER   m = 3, n = 3;
LEAF CELLS   SignExt, FullMult, LSignExt, Comp, RComp, Add;
MAIN
    multiplier = adder | row[n] | (| (row[i] (i = n - 1 .. 1)));
    row[i] = SignExt -- (-- (FullMult (m - 1)));
    row[n] = LSignExt -- (-- (Comp (m - 2))) -- RComp;
    adder = (-- (Add (m + 1))).
```

Lsignext



signext

Figure D-1: SignExt, LSignExt

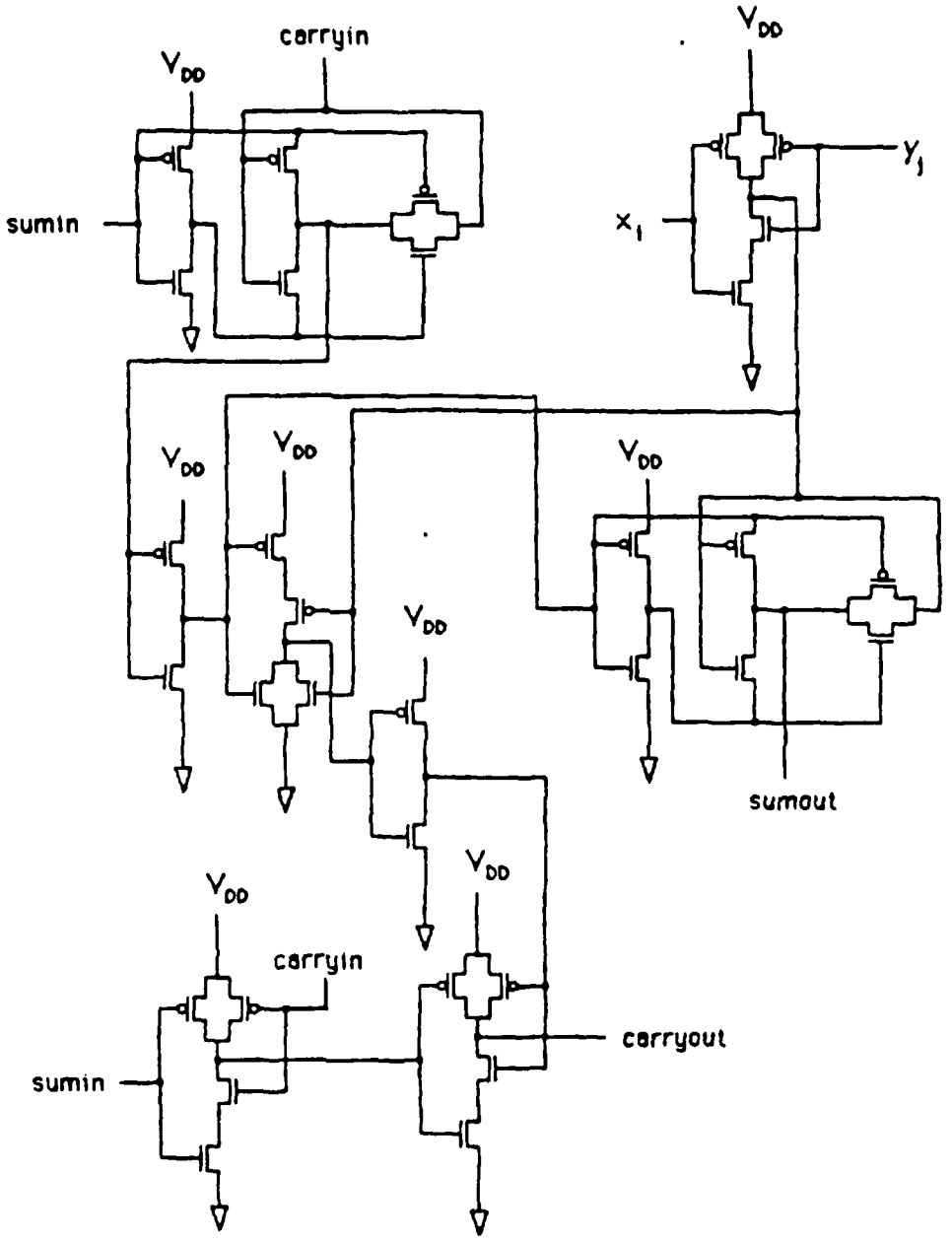


Figure D-2: FullMult

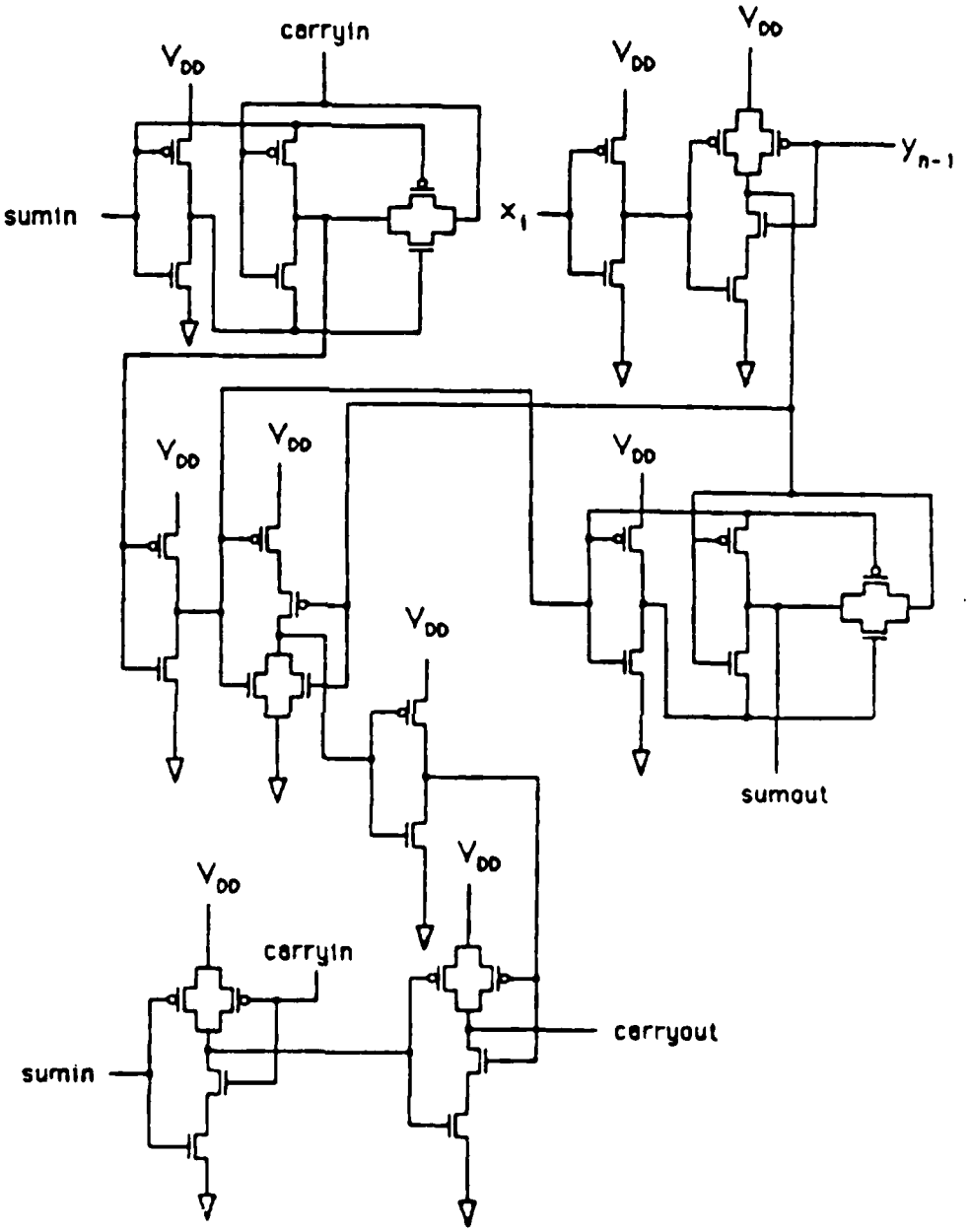


Figure D-3: Comp,RComp

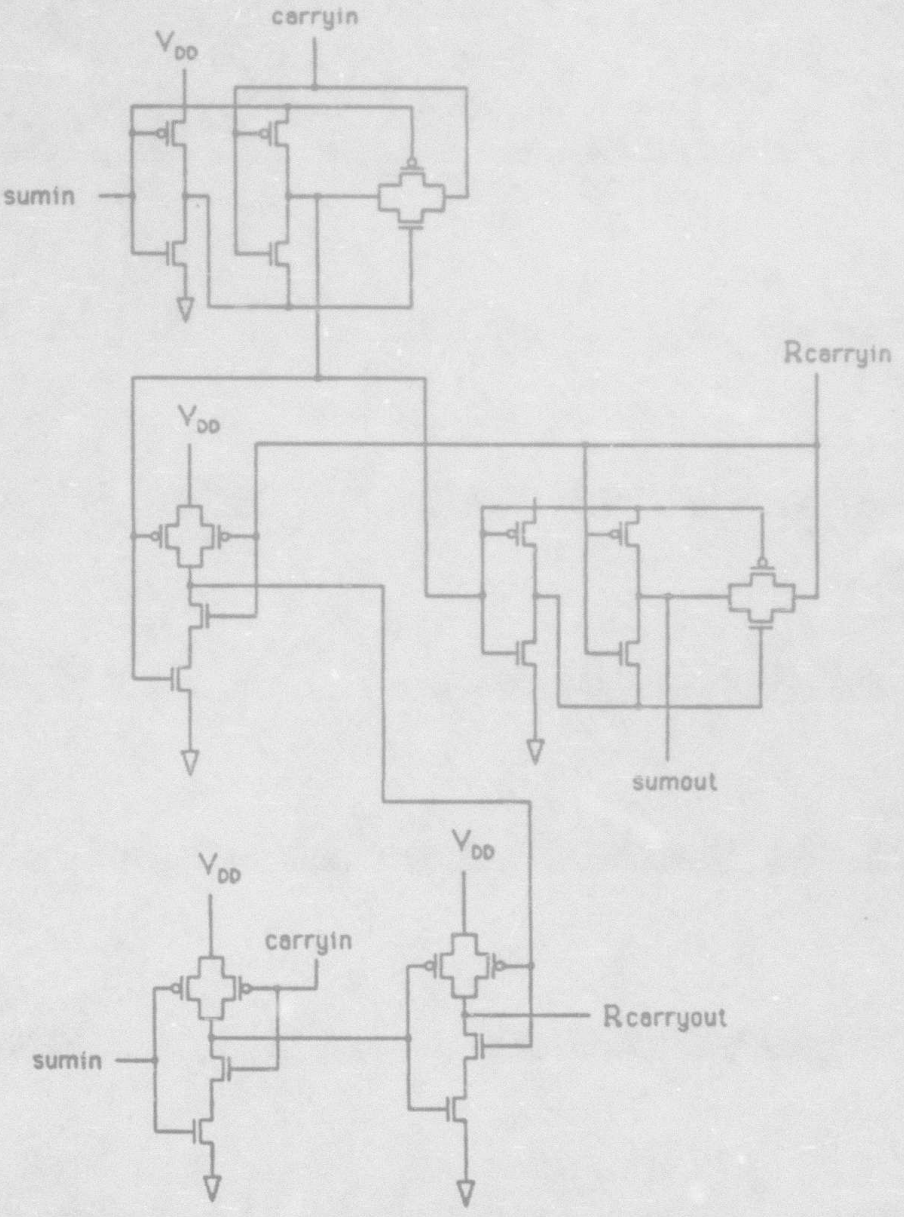


Figure D-4: Add